Head First C

图灵程序设计丛书

嗨翻C语言

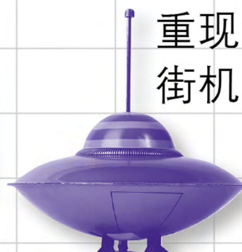
传授C编程专家的
内功心法

用make改变
你的生活

避免尴尬的
指针错误

苏在可变参数
函数的帮助下
变得八面玲珑

重现经典
街机游戏



玩转
C标准库

[美] David Griffiths
Dawn Griffiths 著
程亦超 译

O'REILLY®

人民邮电出版社
POSTS & TELECOM PRESS

数字版权声明

图灵社区的电子书没有采用专有客户端，您可以在任意设备上，用自己喜欢的浏览器和PDF阅读器进行阅读。

但您购买的电子书仅供您个人使用，未经授权，不得进行传播。

我们愿意相信读者具有这样的良知和觉悟，与我们共同保护知识产权。

如果购买者有侵权行为，我们可能对该用户实施包括但不限于关闭该帐号等维权措施，并可能追究法律责任。

TURING 图灵程序设计丛书

嗨翻C语言

[美] David Griffiths
Dawn Griffiths 著
程亦超 译

O'REILLY®

Beijing • Cambridge • Farnham • Köln • Sebastopol • Tokyo

O'Reilly Media, Inc.授权人民邮电出版社出版

人民邮电出版社
北　京

图书在版编目（CIP）数据

嗨翻C语言 /（美）格里菲思（Griffiths, D.），（美）格里菲思（Griffiths, D.）著；程亦超译. —北京：人民邮电出版社，2013.9

（图灵程序设计丛书）

书名原文：Head First C

ISBN 978-7-115-31884-8

I. ①嗨… II. ①格… ②程… III. ①C语言－程序设计 IV. ①TP312

中国版本图书馆CIP数据核字（2013）第137860号

内 容 提 要

本书向读者提供了C语言的完整学习体验。全书分为三个部分：第1章到第4章是基础知识，包括基本语法、指针、字符串、小工具与源文件；第5章到第8章为进阶内容，有结构、联合、数据结构、堆、函数指针、动/静态链接；最后四章是高级主题，内容涵盖了系统调用、进程间通信、网络编程和多线程。每部分结束后还有一个有趣的实验，可以提高读者的实际操作能力。此外，书中还包含大量的图片、示例和代码，有助于读者对于知识的理解和把握。

本书适用于C开发人员以及对C语言感兴趣的初学者。

◆ 著　　[美] David Griffiths，Dawn Griffiths

译　　程亦超

责任编辑　　李松峰

执行编辑　　李　洁

◆ 人民邮电出版社出版发行　　北京市崇文区夕照寺街14号

邮编　100061　　电子邮件　315@ptpress.com.cn

网址　http://www.ptpress.com.cn

北京　　印刷

◆ 开本：880×1230　1/20

印张：31.5

字数：781千字　　2013年9月第1版

印数：1-5 000册　　2013年9月北京第1次印刷

著作权合同登记号　　图字：01-2012-4881 号

定价：99.00元

读者服务热线：(010) 51095186转604　印装质量热线：(010) 67129223

反盗版热线：(010) 67171154

广告经营许可证：京崇工商广字第0021号

版 权 声 明

Copyright©2012 David Griffiths and Dawn Griffiths.

Simplified Chinese Edition, jointly published by O'Reilly Media, Inc. and Posts & Telecom Press, 2013. Authorized translation of the English edition, 2012 O'Reilly Media, Inc., the owner of all rights to publish and sell the same.

All rights reserved including the rights of reproduction in whole or in part in any form.

英文原版由O'Reilly Media, Inc. 出版2012。

简体中文版由人民邮电出版社出版 2013。英文原版的翻译得到O'Reilly Media, Inc.的授权。此简体中文版的出版和销售得到出版权和销售权的所有者 —— O'Reilly Media, Inc.的许可。

版权所有，未得书面许可，本书的任何部分和全部不得以任何形式重制。

O’Reilly Media, Inc.介绍

O’Reilly Media通过图书、杂志、在线服务、调查研究和会议等方式传播创新知识。自1978年开始，O’Reilly一直都是前沿发展的见证者和推动者。超级极客们正在开创着未来，而我们关注真正重要的技术趋势——通过放大那些“细微的信号”来刺激社会对新科技的应用。作为技术社区中活跃的参与者，O’Reilly的发展充满了对创新的倡导、创造和发扬光大。

O’Reilly为软件开发人员带来革命性的“动物书”；创建第一个商业网站（GNN）；组织了影响深远的开放源代码峰会，以至于开源软件运动以此命名；创立了Make杂志，从而成为DIY革命的主要先锋；公司一如既往地通过多种形式缔结信息与人的纽带。O’Reilly的会议和峰会集聚了众多超级极客和高瞻远瞩的商业领袖，共同描绘出开创新产业的革命性思想。作为技术人士获取信息的选择，O’Reilly现在还将先锋专家的知识传递给普通的计算机用户。无论是通过书籍出版，在线服务或者面授课程，每一项O’Reilly的产品都反映了公司不可动摇的理念——信息是激发创新的力量。

业界评论

“O’Reilly Radar博客有口皆碑。”

——Wired

“O’Reilly凭借一系列（真希望当初我也想到了）非凡想法建立了数百万美元的业务。”

——Business 2.0

“O’Reilly Conference是聚集关键思想领袖的绝对典范。”

——CRN

“一本O’Reilly的书就代表一个有用、有前途、需要学习的主题。”

——Irish Times

“Tim是位特立独行的商人，他不光放眼于最长远、最广阔的视野并且切实地按照Yogi Berra的建议去做了：‘如果你在路上遇到岔路口，走小路（岔路）。’回顾过去Tim似乎每一次都选择了小路，而且有几次都是一闪即逝的机会，尽管大路也不错。”

——Linux Journal

谨以此书献给C语言之父Dennis Ritchie（1941~2011）

对Head First丛书的赞誉

“Kathy和Bert的《深入浅出Java》把书本变成了图形界面。作者通过一种诙谐、嬉皮的调调，把学习Java变成了一个充满未知的过程，我总忍不住好奇地想：‘作者接下来会干嘛？’”

——Warren Keuffel，《软件开发杂志》

“《深入浅出Java》用引人入胜的手法带你走进Java世界的大门，书中没有令人望而却步的‘课后习题’，而是设置了很多实践环节。很少有教科书能像这本书一样在做到机智、幽默、嬉皮和实用的同时，还能教会你怎么使用对象序列化和网络发布协议。”

——Dr. Dan Russell，IBM Almaden研究中心用户科学与体验组主任、斯坦福大学人工智能讲师

“《深入浅出Java》单刀直入，玩世不恭，妙趣横生，引人入胜，你一定能从中学到东西！”

——Ken Arnold，前Sun公司高级工程师、《Java编程语言》合著者（另一个作者是Java之父James Gosling）

“举重若轻，犹如把千斤重的书本从我心头卸下。”

——Ward Cunningham，Wiki之父、Hillside Group创始人

“这本书非常适合我们这些喜欢新技术的程序员，它对实际的开发很有参考价值，没有枯燥乏味的‘学究腔’，读罢感到神清气爽。”

——Travis Kalanick，Scour and Red Swoosh Member创始人、MIT TR100会员

“过去世界上有三种书：用来买的书，用来收藏的书，用来放在桌子上的书。直到O’Reilly和Head First团队的出现，世界上有了第四种书——Head First系列的书——满是折角、破损不堪、随身携带的书。我把《深入浅出SQL》放在了触手可及的地方。而且，就连我审稿用的PDF文档也被我莫名其妙地翻坏了。”

——Bill Sawyer，Oracle ATG课程主管

“这本书条理分明、幽默风趣、货真价实，即使不是程序员也能从这本书中学到解决问题的方法。”

——Cory Doctorow，Boing Boing合作编辑、*Down and Out in the Magic Kingdom*及*Someone Comes to Town, Someone Leaves Town*作者

“我一拿到这本书就开始读了起来，欲罢不能，这本书实在太酷了！不仅有趣，涵盖了那么多东西，还抓住了要点，叫人毕生难忘。”

——Erich Gamma，IBM杰出工程师、《设计模式》合作者

“是我读过最有趣也是最具智慧的一本关于软件设计的书。”

——Aaron LaBerge，ESPN.com技术副总监

“过去人们需要反复试验才能学到的东西现在已经浓缩为了一本引人入胜的书。”

——Mike Davidson，Newsvine公司CEO

“每一章都围绕着优雅的设计展开，每一个概念在传达智慧的同时也不失实用。”

——Ken Goldstein，迪士尼在线执行副总裁

“我爱*Head Frist HTML with CSS & XHTML*，它寓教于乐！”

——Sally Applin，UI设计师、艺术家

“过去我在看设计模式的书时总是晕乎乎的，恨不得头悬梁锥刺股，但这本书却让我体会到了学习设计模式的乐趣。”

“当其他书还在老和尚念经时，这本书已经开始高声歌唱：‘摇滚吧，宝贝！’”

——Eric Wuehler

“爱死这本书了，我当着老婆的面吻了它。”

——Satish Kumar

对本书的赞誉

"《嗨翻C语言》可能很快就会被证明是学习C语言的最佳书籍。我觉得它会成为每所大学C语言的标准教材。很多编程书籍因循守旧。不过这本书却使用了完全不同的方式。它将教你如何成为一名真正的C程序员。"

——Dave Kitabjian，NetCarrier Telecom软件开发总监

"《嗨翻C语言》是一本用经典'Head First'的方式轻松介绍C语言的教材。图片、笑话、练习以及实践让读者逐渐并稳固地掌握C语言的基础知识……由此，读者可以进入Posix和Linux系统编程中更高级的技术殿堂。"

——Vince Milner，软件工程师

《嗨翻C语言》的作者

David Griffiths

他12岁时看到一部介绍Seymour Papert工作的纪录片，从此踏上编程之路。15岁那年实现了Papert的LOGO编程语言。大学专业是理论数学，毕业后开始编程，并成为一名专栏作家。现在有三个头衔：敏捷教练、程序员和车库管理员。能够用十多种编程语言编程，但只精通其中的一种。写作、编程、辅导之余，David喜欢和心爱的妻子——也是本书的合著者Dawn一起旅行。

在写《嗨翻C语言》之前，David写过两本Head First系列的书：*Head First Rails*和*Head First Programming*。

你可以在Twitter上“粉”David：

http://twitter.com/dogriffiths。

Dawn Griffiths

在英国一所顶尖的大学开始了她的数学生涯，获得了数学系的荣誉学位，毕业以后投身软件开发行业，迄今已经有15年的IT行业从业经验。

在和David一起写《嗨翻C语言》之前，Dawn曾写过两本Head First系列的书（《深入浅出统计学》和*Head First 2D Geometry*），同时还主持过该系列其他几本书。

除了为Head First系列写书，Dawn对太极拳、跑步、编蕾丝和烹饪也很有研究。她十分享受和丈夫在一起旅行的时光。

译者序

1969年“阿波罗11号”登月成功。贝尔实验室中一个叫Ken Thompson的年轻人为了一圆翱游太空的梦想，在当时的Multics[1]系统上写了一个叫《星际之旅》的游戏。但当时大型机的机时费很贵，每玩一次公司就要为此支付75美金，于是Thompson打起了小型机PDP-7的主意。但当时的PDP-7只有一个简陋的运行时系统，不支持多用户，为了能双人对战，Thompson找来Dennis Ritchie一起开发新的操作系统。

他们只花了一个月的时间就用汇编语言写出了操作系统的原型。同事Peter Neumann看到后，戏称这个系统为Unics[2]。Unix这个名字典出于此。

1971年，第一版的Unix已经能够支持两名用户在PDP-11上玩《星际之旅》了，但因为当时的Unix是用汇编语言写的，无法移植到其他机器上，所以他们决定用高级语言重写Unix，可当时的高级语言无论从运行效率还是功能上都无法满足他们的需要。Thompson先是在BCPL的基础上萃取出了B语言，Ritchie又在B的基础上进行了重新设计，这才有了今天大名鼎鼎的C语言。

而现在你手上的就是一本关于C语言的书。

本书分为三个部分。

- 第1章到第4章是基础知识，包括基本语法、指针、字符串、小工具与源文件。
- 第5章到第8章为进阶内容，有结构、联合、数据结构、堆、函数指针、动/静态链接。
- 最后四章是高级主题，内容涵盖了系统调用、进程间通信、网络编程和多线程。

每部分结束后还用一个实验来提高读者的动手能力。

本书最大的特点是每次在引出新概念前都会先提出一个问题，让读者在知道怎样做（how）之前先知道为什么这么做（why），并在解决问题的过程中不断提出新问题，让读者去解决，从而加深理解；书中还设有很多“问答”环节，提出并回答了一些读者在学习过程中可能会遇到的问题。除此之外，作者还使用了大量拟人手法，例如让编译器化身公众人物在访谈节目中现身说法，抑或让静态库和动态库对簿公堂。谈笑风生间，它们的特点，跃然纸上。无论你是音乐发烧友、推理迷，还是填字游戏爱好者，都可以在这本书中找到吸引你的元素。

两个改变世界的发明起初不过是为了一个游戏，从这个角度看，这本同样趣味十足的《嗨翻C语言》，能否算是对于这种精神的一种延续呢？

---

① Multics全称为MULTiplexed Information and Computing System（多路信息计算系统）是1964年由贝尔实验室、MIT和通用电气共同研发的一套安装在大型机上的多人多任务操作系统。因为工作进度缓慢，贝尔实验室于1969年退出该计划。

② 意思是UNiplexed Information and Computing System（单路信息计算系统），用来影射Multics。

我在翻译的过程中力求真实传达作者的意图，无论是一个技术上的概念还是一段幽默。为了减轻阅读压力，我还将书中部分代码中的字符串也译为了中文，希望不是画蛇添足。

最后，感谢王琛、邱瑀庭等好友提出的建议；感谢作者David Griffiths耐心解答我提出的每一个问题。感谢图灵的李洁、李松峰、傅志红老师以及各位审读老师提供的帮助与支持。

程亦超

2012年12月17日

O'Reilly的其他相关图书

C in a Nutshell

Practical C Programming

C Pocket Reference

Algorithms with C

Secure Programming Cookbook for C and C++

O'Reilly Head First系列的其他图书

Head First Programming

Head First Rails

Head First Java™

Head First Object-Oriented Analysis and Design (OOA&D)

Head First HTML5 Programming

Head First HTML with CSS and XHTML

Head First Design Patterns

Head First Servlets and JSP

Head First EJB

Head First PMP

Head First SQL

Head First Software Development

Head First JavaScript

Head First Ajax

Head First Statistics

Head First 2D Geometry

Head First Algebra

Head First PHP & MySQL

Head First Mobile Web

Head First Web Design

目录（精简版）

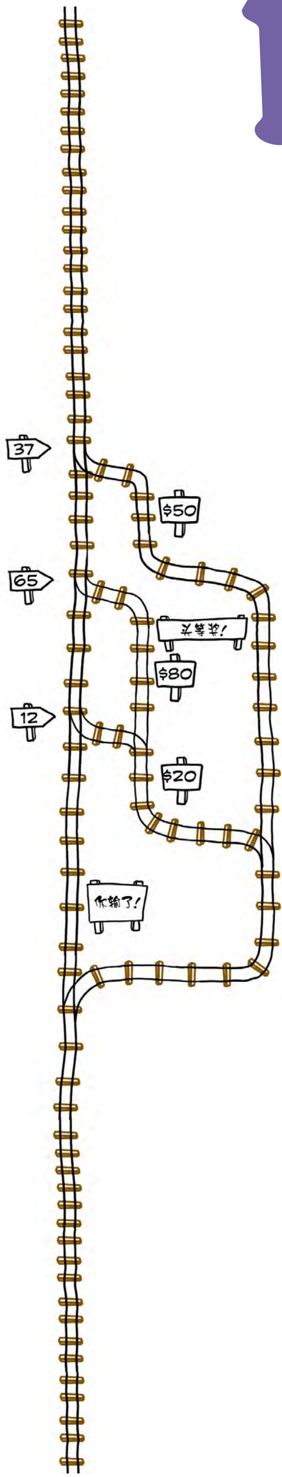
目录（完整版）

引子

让大脑重视C语言。现在你正试着学习某些东西，为了不让学习卡壳，你的大脑也在帮你的忙，大脑在想："最好把空间留给重要的事，比如什么动物是危险的？裸体滑雪是不是一个坏主意？"那么怎么才能欺骗你的大脑，让它认为学好C关系到你下半生的幸福呢？

1 C语言入门

进入C语言的世界



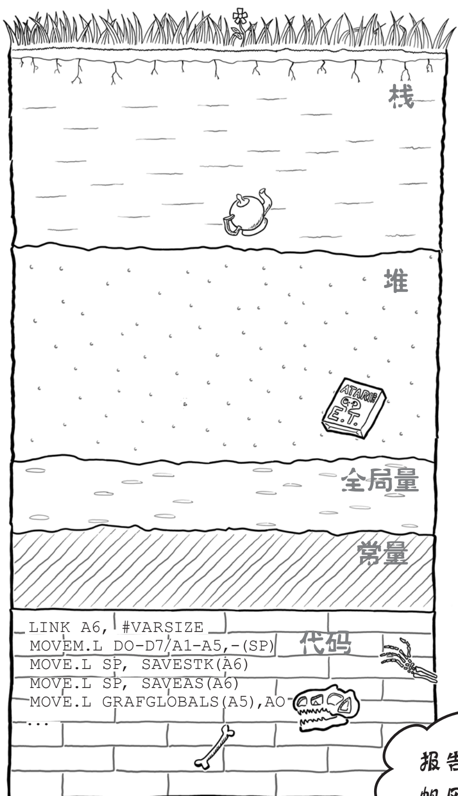
想知道计算机在想什么?

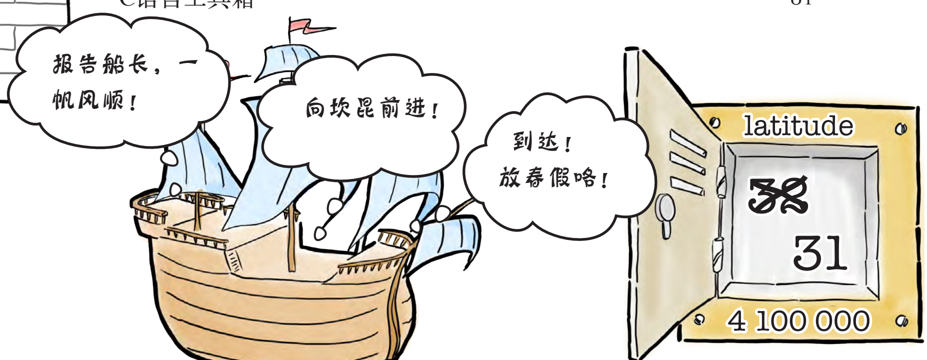
你需要为一款新游戏编写高性能的代码吗？你需要为Arduino编程吗？你需要在iPhone应用中使用高级的第三方库吗？如果是的话，C语言就可以帮上忙了。相比其他大多数语言，C语言的工作层次更低，因此理解C语言可以让你更清楚程序在运行时到底发生了什么，C语言还可以帮助你更好地理解其他语言。来吧，拿起编译器，很快你就能入门了。

2 存储器和指针

指向何方?

如果真的想玩转C语言，就需要理解C语言如何操纵存储器。
C语言在如何使用存储器方面赋予了你更多的掌控权。在本章中，你将揭开存储器神秘的面纱，看到读写变量时到底发生了什么；学习数组的工作原理，以及怎样避免烦人的存储器错误；最重要的是，你将看到掌握指针和存储器寻址对成为一名地道的C程序员来讲有多么重要。





2.5 字符串

字符串原理

字符串不只是用来读取的。

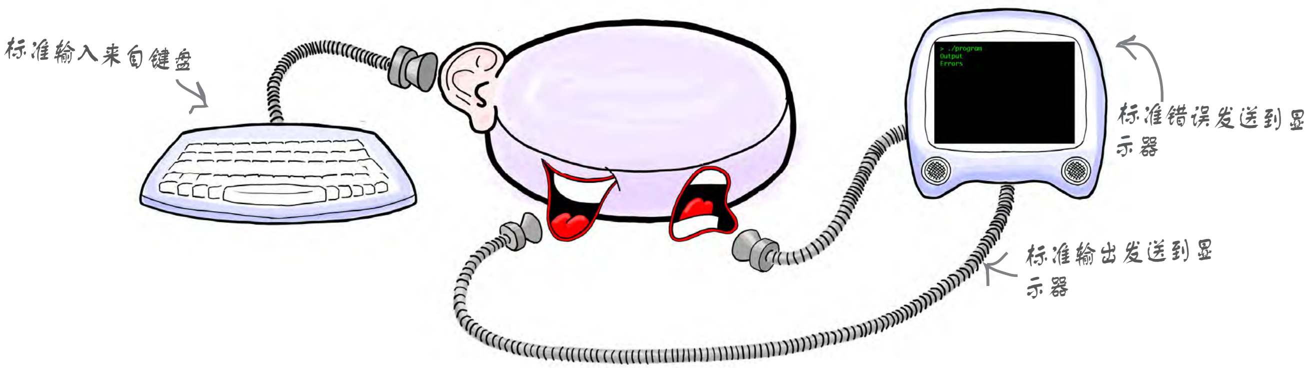
在C语言中字符串其实就是char数组，这你已经知道了，问题是字符串能用来干嘛？该*string.h*出场了。*string.h*是C标准库的一员，它负责处理字符串。如果想要连接、比较或复制字符串，*string.h*中的函数就可以派上用场了。在本章中，你将学会如何创建字符串数组，并近距离观察如何使用**strstr()**函数搜索字符串。

3 创建小工具

做一件事并把它做好

操作系统都有小工具。

C语言小工具执行特定的小任务，例如读写文件、过滤数据。如果想要完成更复杂的任务，可以把多个工具链接在一起。那么如何构建小工具呢？本章中，你会看到创建小工具的基本要素并学会控制命令行选项、操纵信息流、重定向，并很快建立自己的工具。

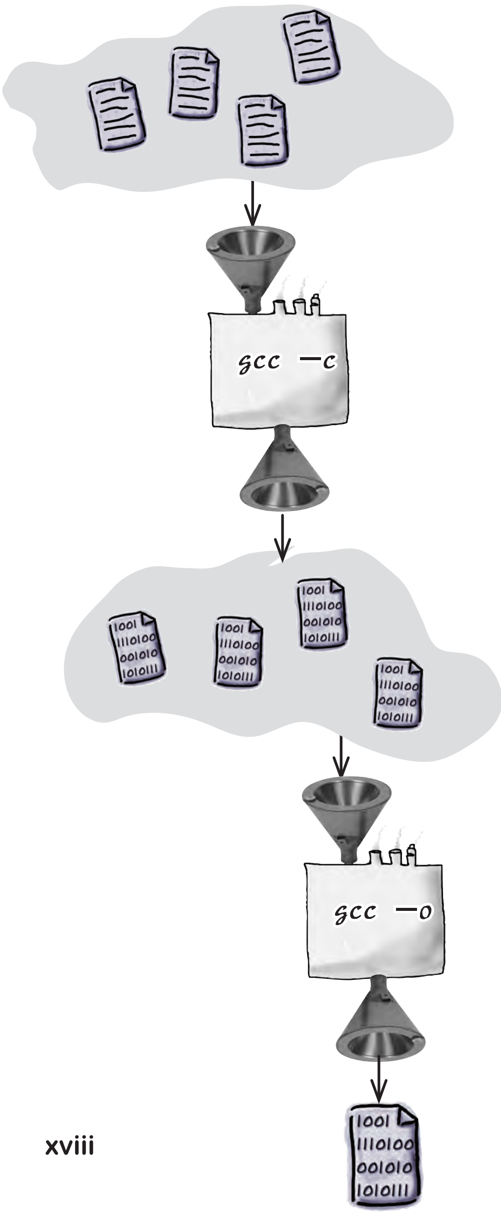


4 使用多个源文件

分而治之

大程序不等于大源文件。

你能想象一个企业级的程序如果只有一个源文件，维护起来有多么困难与耗时吗？在本章中，你将学习怎样把源代码分解为易于管理的小模块，然后把它们合成一个大程序，同时还将了解数据类型的更多细节，并结识一个新朋友：**make**。



C语言实验室 1

Arduino

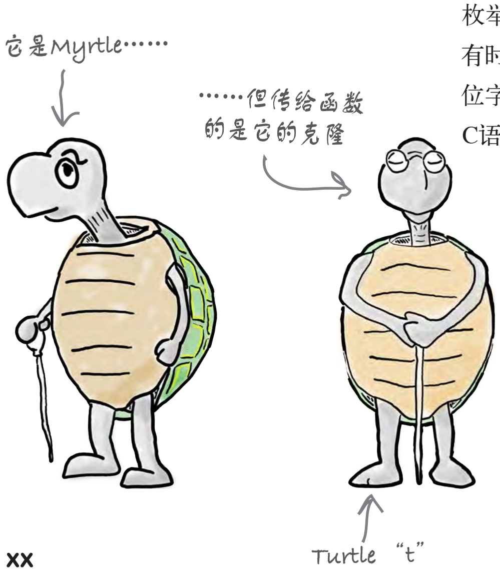
你可曾想过，你的植物告诉你它需要浇水？有了Arduino，植物就可以开口了！本实验中，你将创建一个由Arduino驱动的植物监控器，全用C语言来写。

5 结构、联合与位字段

创建自己的结构

生活可比数字复杂多了。

到目前为止，你只接触过C语言的基本数据类型，但如果想表示数字、文本以外的其他东西呢，或为现实世界中的事物建立模型，怎么办？结构将帮你创建自己的结构，模拟现实世界中错综复杂的事物。在本章中，你将学习如何把基本数据类型组成结构以及用联合处理生活的不确定性。如果你想简单地模拟“是”或“非”，可以用位字段。

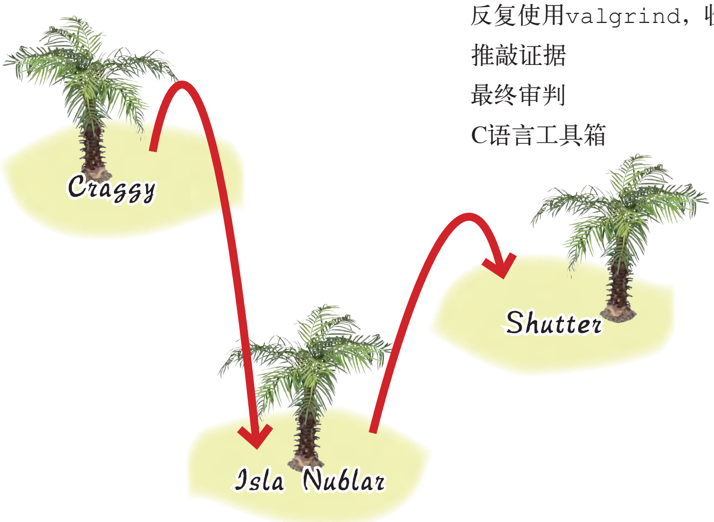


6 数据结构与动态存储

牵线搭桥

一个结构根本不够。

为了模拟复杂的数据需求，通常需要把结构链接在一起。在本章中，你将学习如何用结构指针把自定义数据类型连接成复杂的大型数据结构，将通过创建链表来探索其中的基本原理；同时还将通过在堆上动态地分配空间来学习如何让数据结构处理可变数量的数据，并在完成工作后释放空间；如果你嫌清理工作太麻烦，可以学习一下怎么用valgrind。

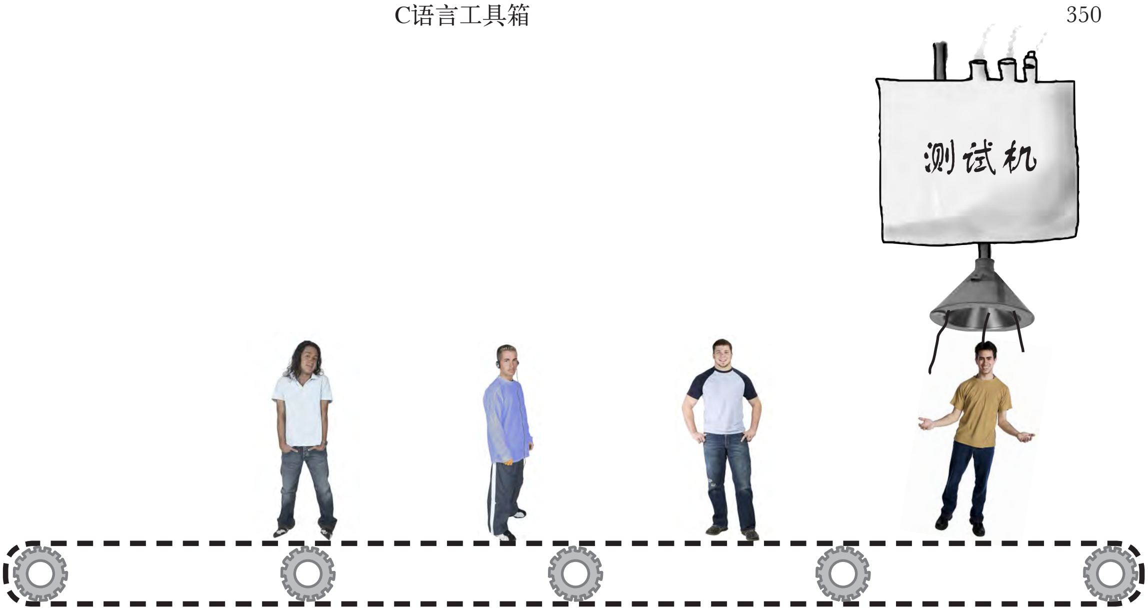


7 高级函数

发挥函数的极限

基本函数很好用，但有时需要更多功能。

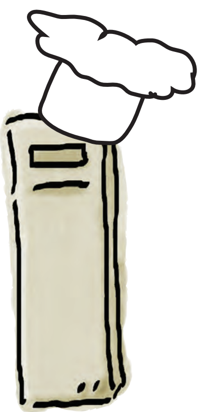
到目前为止，你只关注了一些基本的东西，为了达成目标，需要更多的功能与灵活性。本章你将学习如何把函数作为参数传递，从而提高代码的智商，并学会用比较器函数排序，最后还将学会使用可变参数函数让代码伸缩自如。

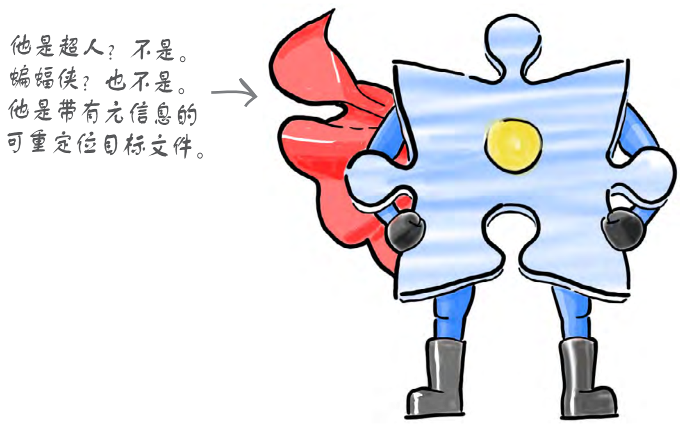


8 静态库与动态库

热插拔代码

你已经见识过标准库的威力。
是时候在代码中发挥这种威力了。在本章中，你将学会创建自己的库，以及在多个程序中复用相同代码；还将掌握编程大师的秘诀——通过动态库在运行时共享代码；最后你将写出易于扩展并可以有效管理的代码。





C语言实验室 2

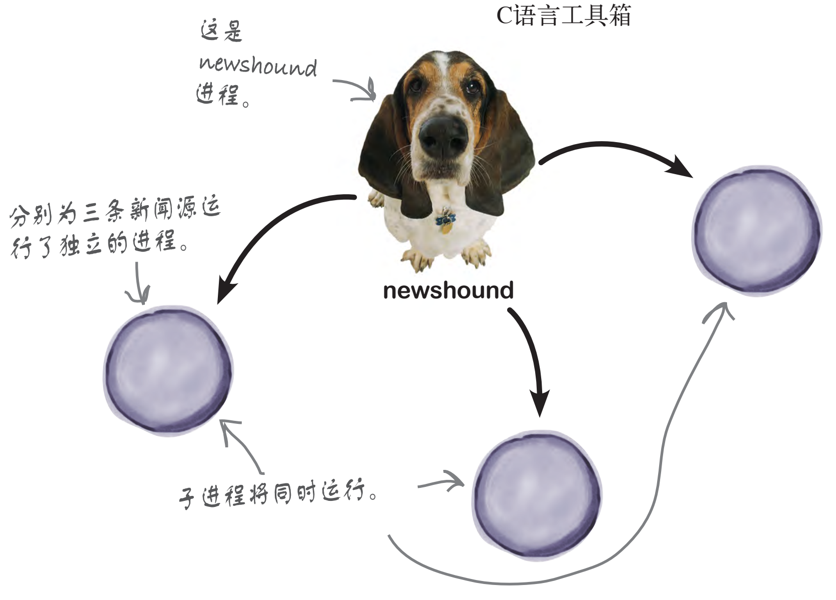
OpenCV

试想一下，当你出门在外，你的计算机能帮你看家，还能让你看到小偷的真面目。在这个实验中，你将借助OpenCV的神奇力量，创建一个基于C语言的入侵者检测器。

9 进程与系统调用

打破疆界

打破常规。
你已经学会了通过在命令行连接小工具的方式建立复杂的程序。但如果你想在代码中使用其他程序怎么办？本章中你将学会如何用系统服务来创建和控制进程，让程序发电子邮件、上网和使用任何已经安装过的程序。本章的最后，你将得到超越C语言的力量。

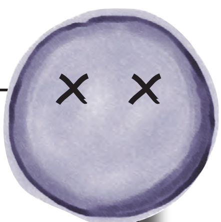


10 进程间通信

沟通的艺术

创建进程只是个开始。

如果你想控制运行中的进程，向进程发送数据或读取它的输出，该怎么办？通过进程间通信，进程可以合力完成某件工作。我们将向你展示如何让程序与系统中其他程序通信，从而提升它的战斗力。



```
#include <stdio.h>

int main()
{
  char name[30];
  printf("Enter your name: ");
  fgets(name, 30, stdin);
  printf("Hello %s\n", name);
  return 0;
}
```

```
File Edit Window Help
> ./greetings
Enter your name: ^C
>
```

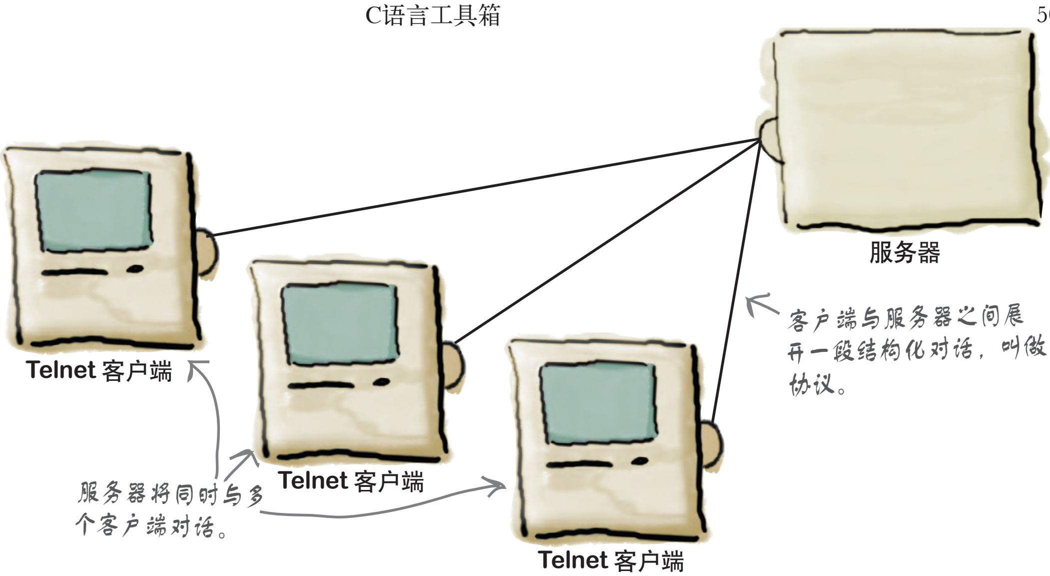
只要按Ctrl-C，程序就会停止运行，为什么会这样？

11 网络与套接字

金窝，银窝，不如127.0.0.1的草窝

不同计算机上的程序需要对话。

你已经学习了怎么用I/O与文件通信，还学习了如何让同一台计算机上的两个进程通信，现在你将走向世界舞台，让C程序通过互联网和世界各地的其他程序通信。本章的最后你将创建具有服务器和客户端功能的程序。

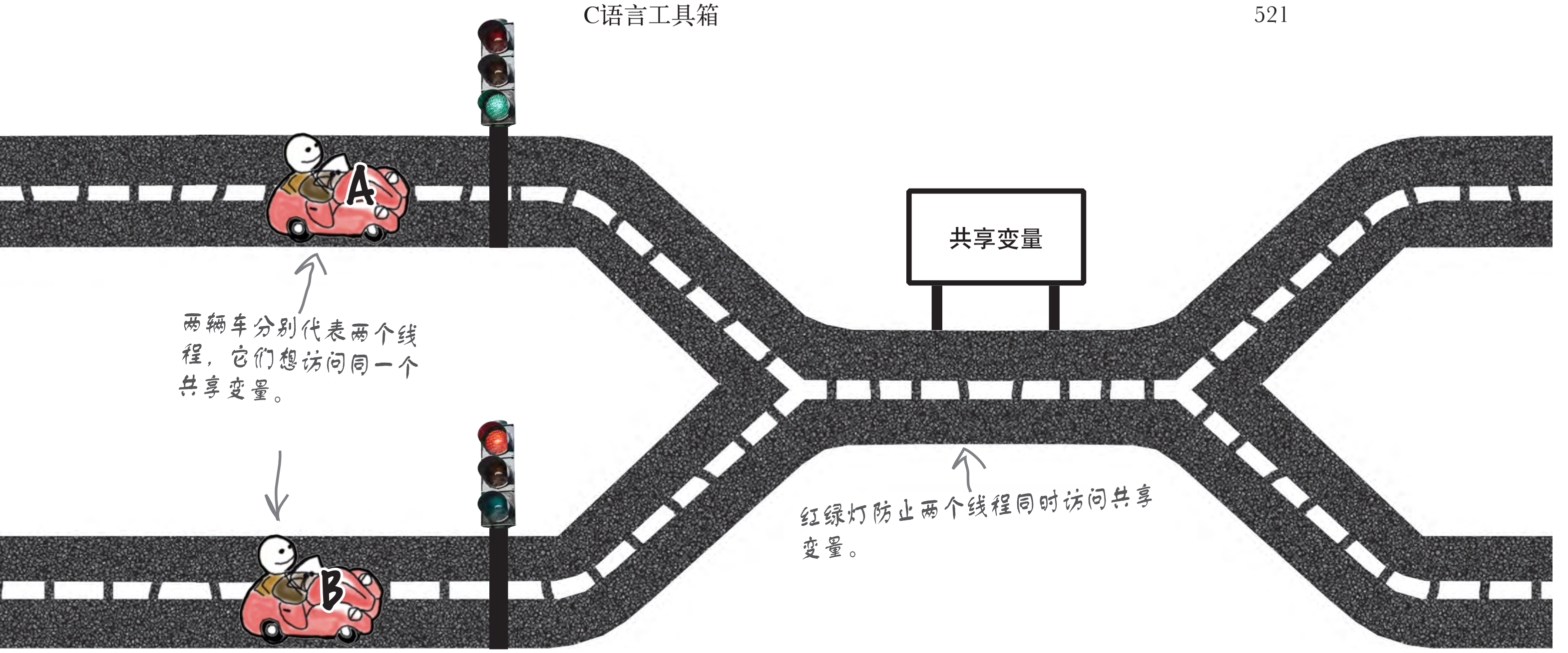


12 线程

平行世界

程序经常需要同时做几件事。

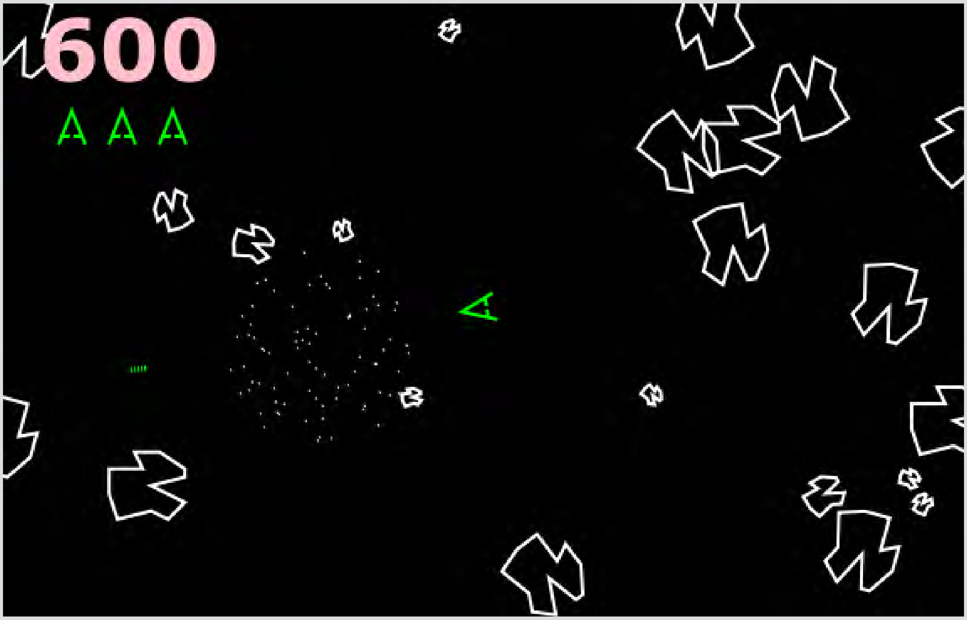
POSIX线程可以派生几段并行执行的代码，从而提高代码的响应速度。但是要小心！线程虽然很强大，但它们之间可能发生冲突。本章你将学习如何用红绿灯来防止代码发生车祸。最终你将学会创建POSIX线程，并使用同步机制来保护共享数据的安全。



C 语言实验室 3

爆破彗星

在本实验中，你将向史上最受欢迎、最长寿的电子游戏——《爆破彗星》——致敬！



饭后甜点

i 十大遗漏知识点

革命尚未成功，同志还需努力。

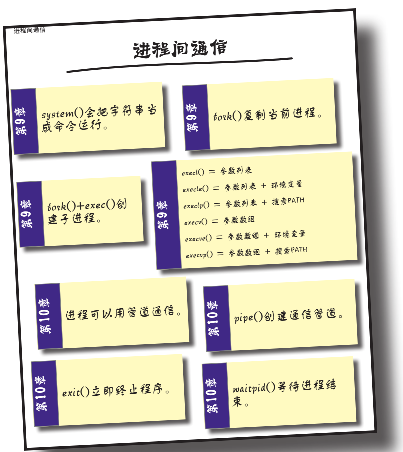
我们认为你还需要知道一些事，如果不讲，总觉得哪里不对劲，但我们又不希望这本书重得只有大力士才提得动，所以我们只做简单介绍。在你放下这本书前，尽情地享用这些“美味佳肴”吧。

话题汇总

ii 总复习

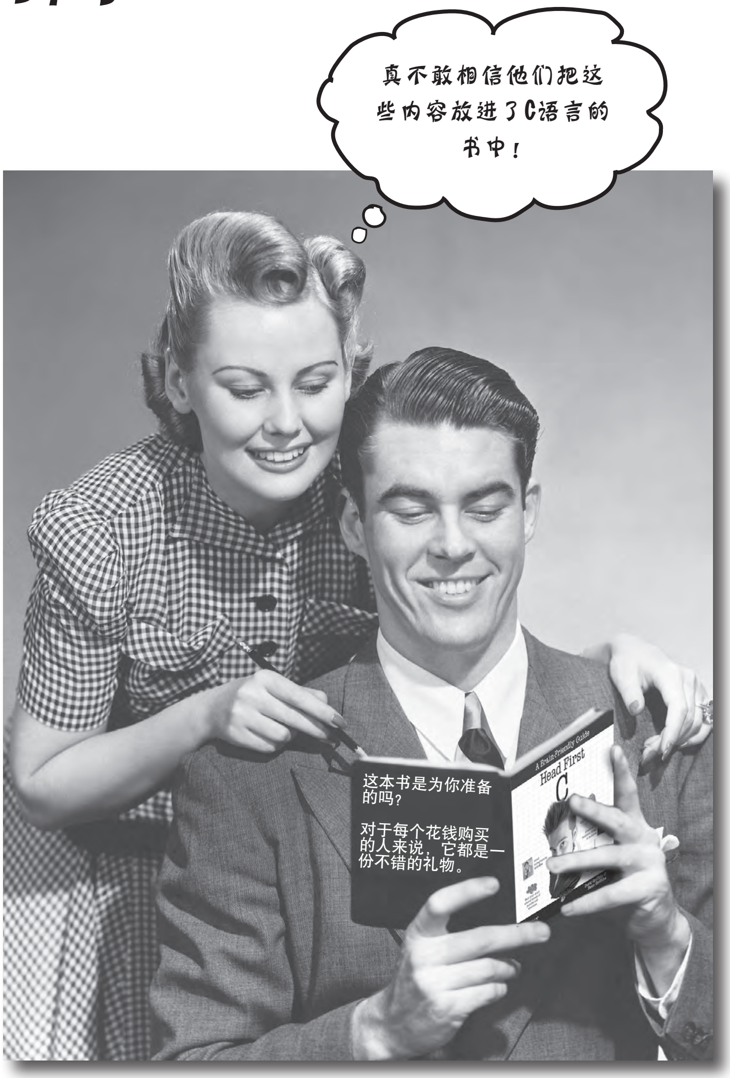
将C语言的特性尽收眼底。

我们把本书中出现过的所有关于C语言的话题和原理都汇总在了这里。把它们过一遍，看你还记得多少。每条话题都标明了来源章节号。如果你想不起来，很容易就能找到原文，甚至还可以把它们剪下来贴在墙上。



如何使用本书

引子



在本节中，我们回答了读者最关心的问题：
“他们为什么要把这些内容放进C语言的书中？”

本书为谁而写？

如果下列问题你都回答“是”：

1. 你会用其他语言编程吗？

2. 你想要掌握C语言，并用它创造软件业的神话，成为亿万富翁，然后在私人小岛上安享晚年吗？

听起来有些遥不可及，但“千里之行，始于足下”，不是吗？

3. 比起枯燥乏味的讲座你更喜欢动手并将所学付诸实践吗？

那么这本书就是为你准备的。

谁与本书无缘？

下列问题只要有一个你回答“是”：

1. 你正在寻找C语言的简介或工具书？

2. 你宁可在大庭广众下和黑猩猩接吻也不愿意吸纳新的知识？你坚信C语言的书应该无所不能而且一定是刻板无趣的吗？

请放下这本书，向后转，前进50步。

市场部友情提示：本书是写给任何人看的，只要你有信用卡就可以购买这本书……我们也收支票。

我们知道你在想什么

“C语言的书怎么可以这么恶搞？”

“那些图片是干嘛的？”

“我真的可以这样学习C语言么？”

我们也知道你的大脑在想什么

大脑渴望新奇的事物，它总是在搜索、扫描和等待不同寻常的东西。大脑生来如此，正是它的这种特性我们才长葆活力。

大脑怎样处理那些老生常谈、平淡无奇的事物呢？它会想尽一切方法阻止它们妨碍自己的真正工作——记住那些重要的事情。大脑不会浪费脑细胞去记忆无聊的事情，它们被“这件事显然不重要”给滤掉了。

大脑又怎么知道哪些事情是重要的呢？假设你去郊游，突然有只老虎跳到你面前，你的大脑和身体会发生哪些反应？

神经元触发、情绪激动、肾上腺素激增。

大脑于是立刻知道……

这些一定很重要！千万别忘记！

但如果你在绝对安全的环境中学习，比如你正在家或图书馆复习迎考，或奉老板之命在一周内掌握某项艰深的技术。

大脑为了帮助你，会阻止那些明显不重要的东西占用稀缺资源。资源应该用来存放真正重要的东西，比如老虎、火灾和“千万别在Facebook上发布自己的裸照”。但你又不能对你的大脑说：“拜托，无论这本书有多无聊，我有多么不情愿，请务必将这些东西记下来。”

我们将Head First的读者视为学习者。

怎样才能学到东西？首先，你必须理解书中内容，然后确保不会忘记。这并不代表填鸭式的死记，根据认知科学、神经生物学和教育心理学的最新研究，学习不仅仅是把书上的文字全部背下来。我们知道如何激活你的大脑，让你有效地学习。

Head First学习守则：

可视化。图片比单纯的文字更容易记忆、学习起来更有效果（知识的回想和转化率可以提高89%）。图片让事情更加容易理解，**把文字放在相关图片的内部或附近**，而不是图片下面或另一页上，学习者解决相关问题的能力将提高两倍。

使用对话式和个性化的语言风格。最近的一项研究发现，相比于传统的授课方式，使用第一人称和对话的形式把内容直接讲给学生听，学生的考试成绩提高了40%。用讲故事取代照本宣科，使用生活化的语言，轻松一点，胜人一筹！你觉得哪个更容易引起你的注意，一场生动有趣的餐会，还是一场严肃的学术讲座？

让读者深入思考。这么说吧，除非你积极刺激自己的神经元，否则你的大脑只是个摆设。为了让读者解决问题、得出结论和形成新的知识，就要让他们充满动力、亲身参与、感到好奇和受到启发。为此，你需要应对一系列挑战、练习、发人深思的提问和活动来刺激你的左右脑和各种感官。

引起并保持读者的注意。人人都有这样的经验：明明想要认真学习，但是一看书就犯困。大脑只会注意与众不同、有趣、怪异、夺人眼球和出人意料的东西。一旦学习一种全新的、有挑战性的技术变得不再枯燥乏味，大脑学习起来就会很快。

打情感牌。你能否记住一件事情和这件事本身的情感色彩有很大关系。你记得你在乎的事情，也记得让你有所感触的事情。我并不是在说忠犬八公和主人之间催人泪下的故事，这里的情感指的是惊讶、好奇、有趣、疑问，以及解决难题后油然而生的成就感，和学会别人不会的技术时那种“舍我其谁”的优越感。

元认知：思考的思考

如果你真心想学习，并且想要学得更快、更深，那你就应该注意你是如何注意的，思考你是如何思考的，学习你是如何学习的。

绝大多数人在成长的过程中没有受过元认知或学习理论方面的教育。我们都知道要学习，却不知该如何学习。

假设你阅读本书的目的是为了学习编程，但又不想花太多时间。如果你想应用你读到的东西，你就要记住它们，为此，你必须先理解它们。为了让这本书（以及其他某本书或任何一段学习经验）的价值最大化，你就要对大脑负责。

秘诀在于让你的大脑认为你正在学习一样很重要的东西，和老虎一样重要，甚至关系到你下半生的幸福。不然，当你在埋首苦读之时，你的大脑却在努力地排斥吸纳新的知识。

如何让大脑将编程视为洪水猛兽？

既有沉闷缓慢的方法，也有快速有效的方式。慢的方法就是不断重复，即使是世界上最乏味的东西，只要反复背它个几百遍，终归能够记住。当你背到第1907遍的时候，大脑说："既然你看了一遍又一遍，姑且认为它很重要吧！"

快的方法是用各种方法**增加大脑活动**，尤其是不同类型的大脑活动。上一页中我们已经提到了几种方法，它们已经被证明是帮助大脑工作的有效方法。例如，研究表明将文字置于它所描述的图片内部（而不是页面中其他的地方，比如标题或正文中），有助于让大脑弄清文字与图片是如何关联的，这会触发更多的神经元。更多的神经元被触发意味着你的大脑更有可能认为它们是重要的事情，也就更有可能记住这些事情。

对话之所以能够帮助学习，是因为人们在对话时为了能接上对方的话，注意力比平时更集中。神奇的是，大脑并不介意这种对话是发生在你与书本之间的。相反，如果行文风格是那种正儿八经的调调，大脑就会以为你正坐在死气沉沉的教室听老师讲解一种二十年前就已经淘汰了的技术，自然打不起精神。

图片和对话只是开始，好戏还在后头……

我们做了什么

图片，大脑喜欢图片而不是文字，对大脑而言，一图胜千言。在图片和文字配合使用时，我们会把文字嵌入图片内部，因为当文字出现在它所描述的图片内部，而不是标题或正文中时，大脑的认知效果更好。

重复，即把同一样东西以不同的方法、媒介、感官体验呈现给读者，这样做的好处是，相同的内容可能被大脑的不同区域所保存，增加了它被记忆的可能性。

用你**意想不到的**方式来使用概念和图片，谁让大脑喜欢猎奇呢？同时，我们在使用图片和概念时会加入一些情感色彩，因为大脑会受情感的左右。让你心有所感的事情更有可能被记住，即便这种感觉不过是些**小幽默**、**小惊讶**或**小趣味**。

个性化、**对话式的表达方式**，当大脑发现你在对话而不是被动地听讲时，注意力会更集中，即便这种对话不是真的。

超过80个**活动**，“光看不练假把势”，比起单纯地阅读，当你在做事情时大脑能学到和记住更多的东西。为了满足绝大多数读者的需要，我们将习题的难度设定在中等偏上，即稍微有一点挑战，但大部分人都有能力完成。

保持**学习方法的多样性**，每个人喜欢不同的学习方法，有人喜欢按部就班，有人喜欢在深入理解细节之前先对整体有所把握，还有的人喜欢直接看例子。不管你属于哪一种，如果以不同的形式展现相同的内容，每个人都将受益。

同时利用你的左脑与右脑，因为越多的脑细胞参与到学习中，你学到和记住东西的可能性就越大，集中注意力的时间也就越长。左右脑轮流上岗，一个工作的时候另一个休息，可以让你在长时间的学习中始终保持高效的状态。

书中写入了含有多种观点的故事和练习，当大脑被迫做出评估和判断时，它便会进入更深层次的学习。

挑战题，比如一些练习题和需要想一想才能回答的问题。大脑只有在运转起来的时候才能学习和记忆。看别人吃饭，自己的肚子是不会饱的！我们能做的只是保证你的力气用对地方，而不是把脑细胞浪费在处理难以理解的例子、糟糕的文法、拗口的术语以及过于简略的表述上。

使用人，在很多故事、例子和图片中都会出现人，因为你是人，所以你的大脑会更加注意人，而不是东西。

沿虚线剪下，贴在冰箱上
时刻提醒自己。

为了驯服大脑，你需要做的事情

俗话说：“师父领进门，修行在个人。”这些小技巧只是开始，你需要倾听你的大脑，找出哪些技巧对你有效，哪些是无用功，一切只有试了才知道！

1. **慢慢来，理解越深，背的就越少。**
   不要走马观花，时常停下来，想一想。看到一个问题，不要直接去翻答案，想象考试时真的碰到这道题该怎么办。大脑想得越深，学习和记忆的效果就越好。

2. **做练习，写笔记。**
   我们设置了很多习题，但如果你不做，就好比请别人代你吃饭，你永远也不会饱。不要盯着题目看，写点什么下来，研究表明，学习时进行一些身体活动可以起到促进效果。

3. **阅读“这里没有蠢问题”单元。**
   每一个都要读！它们不是附录，而是本书的核心内容，千万不要跳过它们。

4. **让阅读本书成为你睡前最后一件事，至少是最后一件有挑战的事。**
   学习的一部分（尤其是将短期记忆转变为长期记忆的那部分）发生在你把书本放下以后。大脑需要一点时间对知识进行消化，如果你在这段时间里又学习了新的东西，可能会把之前学到的忘掉。

5. **大声读出来。**
   朗读会使大脑的另一部分也活跃起来，如果你尝试理解或记忆某件事情，那么应大声地把它说出来。向另一个人解释，你会学得更快，并在讲解的过程中萌生一些新的想法。

6. **多喝水。**
   大脑喜欢在充满水的环境中工作，脱水（等到你感到口渴说明你已经脱水了）会使你的认知功能下降。

7. **倾听大脑。**
   时刻注意你的大脑是否已经过载，当你发现自己读不进去或前读后忘时，就到了该休息一下的时候了。一旦过了这个点，你的学习效率就会大打折扣，并影响你的进度。

8. **心有所感。**
   你要让大脑知道这件事很重要。试着进入故事布置的场景，根据自己的理解为每一张图片添加注释。埋怨一个蹩脚的笑话不好笑，总比没有想法要好。

9. **多写代码！**
   学习C语言只有一种方法：多写代码。这是本书的主旋律。编程是一种技能，掌握它的唯一方法就是练习。为此，我们提供了很多练习的机会：每一章都有一些习题提出问题让你去解决，不要跳过它们——解题也是学习的一部分！实在不会做偷看一下答案也无伤大雅（谁没有提笔忘字的时候呢？），不过一定要在看答案前自己先做一遍。在你进入下一章的学习之前一定要保证上一章的程序能够正确运行。

用户须知

这是一段学习体验，而不是一本工具书。因此我们扫除了你在学习过程中可能会遇到的一切障碍。第一遍阅读时，请从头看起，因为本书对你的知识背景做了一些假设。

我们假设你是C语言的新手，但不是对编程一窍不通。

我们假设你以前写过一些程序，不一定要很多，但至少已经接触过其他语言（比如JavaScript）中的一些基本概念，例如循环、变量。C是一种不怎么“高级”的高级语言，所以如果你一点编程经验都没有，那么在学习这本书之前应该找本别的书来看看，强烈推荐*Head First Programming*。

你需要在电脑上安装C编译器。

这本书中我们使用了`gcc`（GNU编译器套装），它不但功能十分强大，而且还是免费的。你需要确保你的电脑上已经安装了`gcc`。如果你的操作系统是Linux，恭喜你，你已经拥有了`gcc`；如果你使用的是Mac，你需要安装Xcode开发工具，你可以从苹果应用商店或苹果官网下载；如果你使用的是Windows操作系统，有两种选择：一种是Cygwin（*http://www.cygwin.com*），它可以完全模拟UNIX环境，自然也就包括了`gcc`；如果你只是想创建能够在Windows下运行的程序，MinGW（*http://www.mingw.org*）可能更符合你的需要。

书中所有代码都是跨操作系统平台的，我们极力避免写出只能在一种操作系统中才能运行的代码。但在极个别情况中，不同操作系统上的实现可能会略有不同，但我们会指出来。

我们从教你一些C语言的基本概念开始，然后就带你上战场了。

第1章会介绍C语言的基础知识，有了这些东西打底，到第2章时你就能写一些有实际用途、十分有趣的程序了。其余章节会逐步提高你的编程技巧。一眨眼的功夫，你就从一个C语言菜鸟成长为一名武林高手了。

不要跳过任何活动。

习题和活动不是附加题，它们是这本书的核心内容。它们中有的是为了帮助你记忆，有的是为了便于理解，还有一些为了让你学以致用，总之，不要跳过任何习题。

重复是有意的，而且是重要的。

Head First系列与其他技术书的最大不同在于我们希望你真的能够学到东西，而且看完书之后还能记得它们。绝大多数工具书不以记忆为目的，但这本书的核心是学习，为了加强你的记忆，相同的概念可能重复出现好几遍。

例子尽可能简洁。

读者告诉我们在一个200行的例子中寻找2行能说明问题的代码是一件十分头疼的事儿。本书中的绝大部分示例代码都很短，这样你需要学习的部分也就清楚简洁。别指望这些代码经久耐用，它们甚至不是完整的，它们是专门为了学习而写的，因此功能不一定完整。

“脑力风暴”没有答案。

一部分“脑力风暴”练习没有正确答案，另一部分“脑力风暴”练习答案不唯一，你需要心里有数，而在一些练习中，你会找到一些提示，它们将指引你走向胜利之门。

技术审校团队

Dave Kitabjian

Vince Milner

技术审校人员

Dave Kitabjian，电气工程和计算机工程的双学位，拥有20年的咨询、集成、架构经验，曾为多家世界500强公司和高科技创业公司的客户提供信息系统解决方案。工作之余，Dave喜欢弹吉他、练钢琴、陪伴太太和三个孩子。

Vince Milner，干了20多年的C（和其他语言）程序员，在多种平台上工作过。虽然在攻读数学系硕士学位，但下棋时却屡屡输给六岁孩童。

致谢

编辑

首先，要感谢的人是Brian Sawyer，是他让我们写了这本书。Brian在每一个环节都十分信任我们，让我们有足够的自由去试验新的想法，在截稿日期到来的那几天也没有太抓狂。

O'Reilly团队

感谢以下这些人一路以来对我们的帮助：Karen Shaner找图片的本领堪称一绝，有她在事情就好办多了；在波士顿，Laurie Petrycki让我们吃得很好，大大鼓舞了我们的斗志；Brian Jepson带我们走进了Arduino的神奇世界；首发小组制作出这本书的第一版电子版供人下载；最后要感谢Rachel Monaghan和制作小组严格把关，他们是幕后英雄。你们个个都是好样的。

家人、朋友和同事

我们在这次的Head First之旅中认识了很多朋友，感谢Lou Barr、Brett McLaughlin和Sanders Kleinfeld教了我们很多东西！

David：感谢Andy Parker、Joe Broughton、Carl Jacques、Simon Jones和其他朋友，十分抱歉我在忙于写作的这段时间里疏远了你们。

Dawn：要不是亲朋好友们的鼎力支持，这本书可没那么容易写成！特别要感谢爸爸妈妈、Carl、Steve、Gill、Jacqui、Joyce和Paul，真心感谢你们的支持和鼓励！

没有他们就没有这本书

我们的技术审查团队完成了一系列了不起的工作，让我们少走了很多弯路，并确保我们写的东西都是对的。同样感谢那些对首发试读版给予反馈的人，是你们让这本书变得更好。

最后，向这个伟大系列丛书的创始人Kathy Sierra和Bert Bates致谢！

Safari®在线图书

Safari Books Online（www.safaribooksonline.com）是一所按需出版的数字图书馆，它以图书和视频的形式出版世界一流技术、商务作家的专业作品。Safari Books Online是技术专家、软件开发人员、Web设计师、商务和创意人士从事科学研究、解决问题、学习和进行认证培训的主要资源。

Safari Books Online向组织机构、政府机关和个人提供各种产品组合和价格方案。订阅者可通过具有完整搜索功能的数据库获取O'Reilly Media、Prentice Hall Professional、Addison-Wesley Professional、Microsoft Press、Sams、Que、Peachpit Press、Focal Press、Cisco Press、John Wiley & Sons、Syngress、Morgan Kaufmann、IBM Redbooks、Packt、Adobe Press、FT Press、Apress、Manning、New Riders、McGraw-Hill、Jones & Bartlett、Course Technology以及其他几十家出版社的上千种图书、培训视频和预出版书稿。想要了解更多Safari Books Online的信息，请访问我们的网站。

1 C语言入门

进入C语言的世界

想知道计算机在想什么吗？

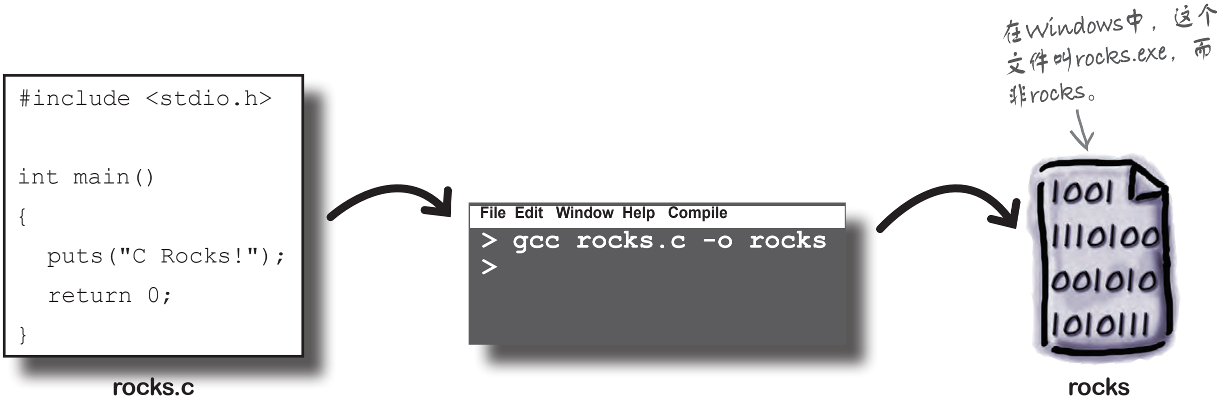
你需要为一款新游戏编写高性能的代码吗？你需要为Arduino编程吗？你需要在iPhone应用中使用高级的第三方库吗？如果是的话，C语言就可以帮上忙了。相比其他大多数语言，C语言的工作层次更低，因此理解C语言可以让你更清楚程序在运行时到底发生了什么，C语言还可以帮助你更好地理解其他语言。来吧，拿起编译器，很快你就能入门了。

C语言用来创建空间小、速度快的程序

C语言旨在创建空间小、速度快的程序。它比其他大多数语言的抽象层次更低，也就是说用C语言写的代码更加接近机器语言。

C语言的工作方式

计算机只理解一种语言——机器代码，即一串二进制0、1流。你可以在编译器的帮助下将C代码转化为机器代码。



1 源代码

从创建一个源文件开始，源文件中是供人阅读的C代码。

2 编译

通过编译器运行源代码，编译器会检查错误，一旦它觉得没问题，就会编译源代码。

3 输出

编译器会创建一个叫可执行文件的新文件，文件中是机器代码，即计算机能够理解的0、1流，而这个文件就是可以运行的程序。

为了写出速度快、空间小、可移植性高的程序，人们常使用C语言。绝大多数的操作系统、其他计算机语言和大多数游戏软件都是用C语言写的。

> 你可能会遇到三种C标准。ANSI C始于20世纪80年代后期，适用于最古老的代码；1999年开始的C99标准有了很大的改进；在2011年发布的最新标准C11中，加入了一些很酷、很新的语言特性。不同版本的标准之间差别不是很大，如果碰到我们会指出。

磨笔上阵

猜一猜这些代码片段分别会做什么。

描述一下你认为这些代码会做什么。

```
int card_count = 11;
if (card_count > 10)
    puts("这副牌赢面很大，我要加注！");
```

```
int c = 10;
while (c > 0) {
    puts("我决不在课堂上写代码！");
    c = c - 1;
}
```

```
/* 假设人名小于20个字符。 */
char ex[20];
puts("输入男友的名字：");
scanf("%19s", ex);
printf("亲爱的%s，我们分手吧。\n", ex);
```

```
char suit = 'H';
switch(suit) {
case 'C':
    puts("梅花(Clubs)");
    break;
case 'D':
    puts("方块(Diamonds)");
    break;
case 'H':
    puts("红心(Hearts)");
    break;
default:
    puts("黑桃(Spades)");
}
```

磨笔上阵解答

即使你还不能全部理解，也不用担心，稍后我们会详细地解释这里的每样东西。

```
int card_count = 11;
if (card_count > 10)
    puts("这副牌赢面很大，我要加注！");
```

整型（integer）是一个整数。

在命令提示符或终端显示字符串。

创建整型变量并将其设为11。

计数超过10？

是的话就在命令提示符显示消息。

```
int c = 10;
while (c > 0) {
    puts("我决不在课堂上写代码！");
    c = c - 1;
}
```

花括号定义了块语句。

创建整型变量并将其设为10。

只要值为正……

……显示消息……

……计数减小。

这是一段需要重复执行的代码的尾部。

```
/* 假设人名小于20个字符。 */
char ex[20];
puts("输入男友的名字： ");
scanf("%19s", ex);
printf("亲爱的%s，我们分手吧。\n", ex);
```

表示“将用户输入的任意字符保存在ex数组中”。

在此处插入这个字符串代替%s。

这是注释。

创建一个有20个字符的数组。

在屏幕上显示消息。

将用户输入的内容保存在数组中。

显示一条包含输入文本的消息。

```
char suit = 'H';
switch(suit) {
case 'C':
    puts("梅花(Clubs)");
    break;
case 'D':
    puts("方块(Diamonds)");
    break;
case 'H':
    puts("红心(Hearts)");
    break;
default:
    puts("黑桃(Spades)");
}
```

switch语句检查一个变量的不同取值。

创建字符变量，保存字母“H”。

查看变量的值。

是“C”吗？

是则显示“梅花（Clubs）”这个词。

然后跳过其他检查。

是“D”吗？

是则显示“方块（Diamonds）”这个词。

然后跳过其他检查。

是“H”吗？

是则显示“红心（Hearts）”这个词。

然后跳过其他检查。

否则……

显示“黑桃（Spades）”这个词。

检测到此结束。

完整的C程序长啥样？

为了创建完整的程序，需要在C源文件中输入代码。任何文本编辑器都可以创建C源文件，它们的文件名通常以.*c*结尾。← 虽然这只是惯例，但应该遵守。

我们来看一个典型的C源文件。

1 C程序通常以注释开头。
注释描述了文件中这段代码的意图，也可能包含一些许可证或者版权信息。在这个地方（或文件的任何地方）添加注释不是必需的，但加上是个好的做法，也是大多数C程序员希望看到的。

注释以/*开始。

这些*号可加可不加，这里加上它们只是为了好看。

注释以*/结尾。

```
/*
 * 计算牌盒中牌数量的程序。
 * 本代码使用“拉斯维加斯公共许可证”。
 * (c)2014，学院21点扑克游戏小组。
 */
#include <stdio.h>

int main()
{
    int decks;
    puts("输入有几副牌");
    scanf("%i", &decks);
    if (decks < 1) {
        puts("无效的副数");
        return 1;
    }
    printf("一共有%i张牌\n", (decks * 52));
    return 0;
}
```

2 接下去是include部分。
C语言是一种很小的语言，如果不使用外部库，它几乎什么也干不了。为了告诉编译器程序要使用哪些外部代码，需要包含（include）相关库的头文件。*stdio.h*是最常见的头文件，stdio库中包含了那些能在终端读写数据的代码。

3 在源文件中找到的最后一样东西是函数。
所有的C代码都在函数中运行。对任何C程序来讲，最重要的函数是**main()**函数。main()函数是程序中所有代码的起点。

让我们仔细研究一下main()函数。

main()函数聚焦

计算机会从main()函数[1]开始运行程序。它的名字很重要：如果没有一个叫main()的函数，程序就无法启动。

main()函数的**返回类型**是int。这是什么意思呢？当计算机在运行程序时，它需要一些方法来判断程序是否运行成功，计算机正是通过检查main()函数的返回值来做到这一点。如果让main()函数返回0，就表明程序运行成功；如果让它返回其他值，就表示程序在运行时出了问题。

这是返回类型：main函数的返回类型必须是int。

因为这个函数叫“main”，程序将从这里开始运行。

只要有参数，就应该在这里提到它们。

函数体总是被花括号包围。

```
int main()
{
    int decks;
    puts("输入有几副牌");
    scanf("%i", &decks);
    if (decks < 1) {
        puts("无效的副数");
        return 1;
    }
    printf("一共有%i张牌\n", (decks * 52));
    return 0;
}
```

函数名在返回类型之后出现，如果函数有参数，可以跟在函数名后面。最后是函数体，函数体必须被花括号包围。

百宝箱

printf()函数用于显示格式化输出，它用变量的值来替换格式符，像这样：

将第一个参数作为字符串插到这里。

第一个参数。

```
printf("%s说计数是%i", "阿星", 21);
```

将第二个参数作为整型插到这里。

第二个参数。

当调用printf()时，可以包含任意数量的参数，但确保每个参数都要有一个对应的%格式符。

如果想检查程序的退出状态，可以在Windows命令提示符中输入：

```
echo %ErrorLevel%
```

或在Linux或Mac终端中输入：

```
echo $?
```

[1] 在早期的ANSI C标准中，main()函数可以是void类型。但是在C99标准中main函数的返回类型必须是int。——译者注

代码冰箱贴

学院21点扑克游戏小组的队员写了一些代码贴在寝室的冰箱上，但有人把冰箱贴弄乱了！你能用这些冰箱贴重组代码吗？

```
/*
 * 计算牌面点数的程序。
 * 使用“拉斯维加斯公开许可证”。
 * (c)2014学院21点扑克游戏小组。
 */

.........................................................................

.........................................................................

................. main()
{
    char card_name[3];
    puts("输入牌名: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {

      ...............................................................

    } else if (card_name[0] == .................) {
        val = 10;

    } ...................... (card_name[0] == .........) {

      ...............................................................
    } else {
        val = atoi(card_name);
    }
    printf("这张牌的点数是: %i\n", val);

   ...................... 0;
}
```

输入两个字符作为牌名。（指向 scanf("%2s", card_name);）

将文本转为数值。（指向 atoi(card_name)）

冰箱贴：

- `<stdlib.h>`
- `;`
- `;`
- `val = 11`
- `int`
- `'J'`
- `#include`
- `'A'`
- `return`
- `else`
- `#include`
- `if`
- `val = 10`
- `<stdio.h>`

代码冰箱贴解答

学院21点扑克游戏小组的队员写了一些代码贴在寝室的冰箱上，但有人把冰箱贴弄乱了！请用这些冰箱贴重组代码。

```
/*
 * 计算牌面点数的程序。
 * 使用“拉斯维加斯公开许可证”。
 * (c)2014学院21点扑克游戏小组。
 */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    puts("输入牌名: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    printf("这张牌的点数是: %i\n", val);
    return 0;
}
```

这里没有蠢问题

问： card_name[0]是什么意思？

答： 它是用户输入的第一个字符。如果用户输入了10，那么card_name[0]就将是1。

问： 总是得用/*和*/写注释吗？

答： 如果你的编译器支持C99标准，就可以用//开始注释。编译器会将这一行的其余部分当做注释处理。

问： 怎么才能知道我的编译器支持哪种标准？

答： 你可以查看编译器的文档。对gcc来讲，ANSI C、C99和C11这三种标准它全部支持。

如何运行程序？

C语言是一种编译型语言，也就是说计算机不会直接解释代码，而是需要将给人阅读的源代码转化（或编译）为机器能够理解的机器代码，这样计算机才能够执行。

为了编译代码，需要一个叫**编译器**的程序。GNU编译器套件（GNU Compiler Collection），也叫gcc，是最流行的C编译器之一。gcc可以在很多操作系统中使用，而且除了C语言，它还可以编译很多其他语言，最重要的是，它是完全免费的。

下面是用gcc编译并运行程序的过程：

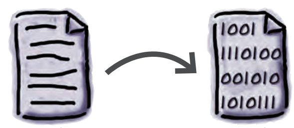
1. 将前一页那道“代码冰箱贴”练习中的代码保存在一个叫cards.c的文件中。

C源代码文件通常以.c结尾。

2. 在命令提示符或终端中使用**gcc cards.c -o cards**命令进行编译。

```
File  Edit  Window  Help  Compile
> gcc cards.c -o cards
>
```

将cards.c编译为一个叫cards的文件。



如果在Windows中，这个文件将是cards.exe。

3. 在Windows命令提示符中输入**cards**或在Mac和Linux终端中输入**./cards**运行程序。

```
File  Edit  Window  Help  Compile
> ./cards
输入牌名：
```

百宝箱

在大部分机器中，可以用下面这个技巧来编译并运行代码：

这里的&&表示：如果成功，就做……

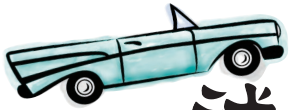
```
gcc zork.c -o zork && ./zork
```

在Windows中，应该输入zork而不是./zork。

这条命令只有在编译成功的情况下才会运行新程序，一旦编译过程中出了问题，它就会跳过运行程序这一步，仅仅在屏幕上显示错误消息。

动手吧！

现在就应该创建cards.c文件，然后编译它。随着本章内容的展开，我们会在它的基础上逐步改进。



试驾

让我们来看看程序能否成功编译和运行。在你的机器上打开命令提示符或终端，试试吧！

这行命令编译了代码并创建了cards程序。

运行命令运行了程序。如果你在Windows中，别加./。

再次运行程序。

用户输入了牌的名字……

……然后程序显示了这张牌的点数。

别忘了：可以把编译和运行合成一步（忘了怎么弄？翻回上一页看看。）。

```
File Edit Window Help 21
> gcc cards.c -o cards
> ./cards
输入牌名:
Q
这张牌的点数是: 10
> ./cards
输入牌名:
A
这张牌的点数是: 11
> ./cards
输入牌名:
7
这张牌的点数是: 7
```

程序工作了！

恭喜！你已经成功编译并运行了C程序。gcc编译器从*cards.c*中提取出了供人阅读的源代码，并将其转换为cards程序中机器才能理解的机器代码。如果你用的是Mac或Linux，计算机会在一个叫cards的文件中创建机器代码；而在Windows中，所有程序的扩展名必须是*.exe*，因此这个文件叫*cards.exe*。

这里没有蠢问题

问： 为什么我在Linux和Mac中运行程序时必须在程序前加上./?

答： 因为在类Unix操作系统中，运行程序必须指定程序所在的目录，除非程序的目录已经列在了PATH环境变量中。

等等，我还是不明白，我们让用户输入牌名时使用了字符数组，为什么要用字符数组？为什么不用字符串（string）或其他东西？

C语言不支持现成的字符串。

但有很多C语言的扩展库提供字符串。

C语言比其他大多数语言的抽象层次更低，因此它不提供字符串，而是用了相似的东西来代替：以字符为元素的数组。如果你用过其他语言，一定已经见过数组了，数组就是一张有名有姓的事物清单，所以`card_name`只是一个变量名，用来引用你在命令提示符输入的那张字符列表的。把`card_name`定义为大小为2个字符的数组，就可以用`card_name[0]`和`card_name[1]`分别引用第一和第二个字符。为了理解字符串的工作原理，让我们深入计算机的存储器，看看C语言是如何处理文本的……

字符串聚焦

字符串只是字符数组，当C语言看到一个这样的字符串时：

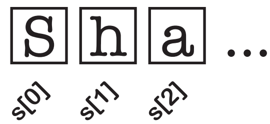
```
s = "Shatner"
```

会把它当做一个数组读取，而这个数组是由一个个独立的字符组成的：

在C语言中，就是这样定义数组的。

```
s = {'S', 'h', 'a', 't', 'n', 'e', 'r'}
```

字符串中的每个字符是数组中的一个元素，这就是为什么可以通过索引来引用字符串中的某个字符，比如s[0]、s[1]。



别在字符串的尽头掉下去

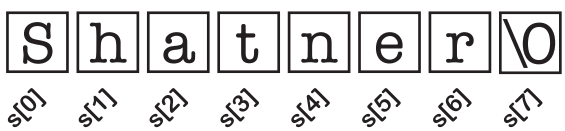
当C语言想要读取字符串中的内容时，会发生什么呢？比如说它想打印字符串吧。在如今的很多语言中，计算机会时刻记录数组的大小，但C语言比大多数语言更低层，它无法确切地知道数组有多长，如果C语言想在屏幕上显示字符串，它就需要知道什么时候会到达字符数组的尾部，为此C语言加入了哨兵字符。

当C语言看到\0就知道该停了。

哨兵字符是一个出现在字符串末尾的附加字符，它的值为\0。每当计算机需要读取字符串的内容时，它会逐一扫描字符数组中的所有元素，直到碰到\0，也就是说当计算机看到下面这个字符串时：

```
s = "Shatner"
```

存储器中实际保存的是：



\0是ASCII字符，它的值为0。

C程序员通常叫它空（NULL）字符。

这就是为什么我们要在代码中像这样定义card_name变量：

```
char card_name[3];
```

字符串card_name只需要记录1到2个字符，但因为字符串要以哨兵字符结尾，所以我们必须把数组的大小定义为3，以放下一个额外的字符。

这里没有蠢问题

问：为什么字符要从0开始编号？为什么不是1？

答： 字符的索引值是一个偏移量：它表示当前要引用的这个字符到数组中第一个字符之间有多少字符。

问：为什么要这样做？

答： 计算机在存储器中以连续字节的形式保存字符，并利用索引计算出字符在存储器中的位置。如果计算机知道c[0]位于存储器1 000 000号单元，那么就可以很快地计算出c[96]在1 000 000 + 96号单元。

问：为什么要设立哨兵字符？难道计算机就不知道字符串的长度吗？

答： 通常不知道。记录数组的长度不是C语言的强项，字符串其实就是个数组。

问：C语言居然不知道数组有多长？

答： 是的，虽然编译器有时可以通过分析代码计算出数组的长度，但一般情况下，C语言希望你来记录数组的长度。

问：单、双引号有区别吗？

答： 有区别，单引号通常用来表示单个字符，而双引号通常用来表示字符串。

问：我应该用双引号（"）定义字符串，还是以显式字符数组的形式定义字符串？

答： 通常应该用双引号来定义字符串。用双引号定义的字符串叫字符串字面值（string literal），比起字符数组，它输入起来也更方便。

问：字符串字面值和字符数组有没有区别？

答： 只有一个区别：字符串字面值是常量。

问：那是什么意思？

答： 也就是说这些字符一旦创建完毕，就不能再修改它们。

问：如果我改了会怎么样？

答： 这取决于编译器，gcc通常会显示总线错误（bus error）。

问：总线错误？那是什么东西？

答： C语言采取不同的方式在存储器中保存字符串字面值。总线错误意味着程序无法更新那一块存储器空间。

虎口拔牙

“等号”不一定表示等于。

在C语言中，“等号”（=）用来赋值（assignment），而“双等号”用来检查两个值是否相等。

把teeth的值设为4

```
teeth = 4;

teeth == 4;
```

检查teeth的值是不是4

如果想要增加或减小变量的值，可以用+=和-=这两个赋值运算符，它们让代码看起来更简短。

teeth加2

```
teeth += 2;

teeth -= 2;
```

teeth减2

最后，如果想要对变量的值加1或减1，可以用++和--。

```
teeth++;   // 加1

teeth--;   // 减1
```

两类命令

到目前为止你看到的所有命令都可以分为以下两类。

做事情

C语言中大部分命令都是语句。简单的语句是一些动作，它们做事情，或告诉我们事情。你已经见过定义变量的语句、从键盘读取输入的语句以及向屏幕显示数据的语句。

```
split_hand();
```

这是一条简单的语句。

当把很多语句组合在一起，就创建出了块语句。块语句是由花括号围起来的一组命令。

```
{
    deal_first_card();
    deal_second_card();
    cards_in_hand = 2;
}
```

这些命令被花括号包围，因此形成了块语句。

只有条件为真才去做事情

例如if这样的控制语句在运行代码之前会检查条件：

```
if (value_of_hand <= 16)
    hit();
else
    stand();
```

这是条件。

如果条件为真，就执行这条语句。

如果条件为假，就执行这条语句。

当条件为真时，if语句一般要做好几件事情，因此if语句通常和块语句一起使用：

```
if (dealer_card == 6) {
    double_down();
    hit();
}
```

如果条件为真，这两条命令都会运行。它们组合进了一条块语句中。

戴还是不戴？

块语句能像处理一条语句那样处理一批语句。在C语言中，if条件语句如下：

```
if (countdown == 0)
  do_this_thing();
```

这条if条件语句运行了一条语句，如果想要在if中运行多条语句呢？只要用花括号把这些语句包起来就行了，C语言会把它们当做一条语句处理：

```
if (x == 2) {
  call_whitehouse();
  sell_oil();
  x = 0;
}
```

C程序员喜欢保持代码的简洁，因此大多数人会省略if条件语句和while循环语句中的花括号，比起写：

```
if (x == 2) {
    puts("Do something");
}
```

大多数C程序员更喜欢写成：

```
if (x == 2)
    puts("Do something");
```

到目前为止的代码

```
/*
 * 计算牌面点数的程序。
 * 使用“拉斯维加斯公开许可证”。
 * 学院21点扑克游戏小组。
 */
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    puts("输入牌名: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    printf("这张牌的点数是: %i\n", val);
    return 0;
}
```

我在想，程序会检查点数的范围吗？如果会就方便了……

Las Vegas Today

专栏广告

教你如何致富

刘富强21点扑克技能专修学校

嗨！最近还好吗？你是个聪明人，我看得出来，因为我也是聪明人，只有聪明人才能发现聪明人，不是吗？听着，我正在干一件低风险高回报的事情，我想让你加入，为什么？谁让我是个好人呢！看清楚了，我是算牌专家，赌神高进正是在下。你问我什么是算牌？好吧，对我来说，算牌是一种事业。

说真的，算牌是一种提高21点胜率的方法。在21点这种扑克游戏中，如果牌盒中高点数的牌足够多，胜利的天平就会倒向玩家——也就是你！

算牌可以帮助你记录还剩几张高点数牌。假设开始计数为0，发牌员发给你的第一张牌是Q，这是一张大牌，因此这副牌中就少了一张高点数的牌，于是你将计数减1：

是一张Q → count – 1

但如果是4那样的小牌，计数就加1：

是一张4 → count + 1

大牌有10和人头牌（J、Q和K），小牌有3、4、5和6。

对每一张大牌和小牌都如此操作，直到计数器变得很大，你就把现金押到下一盘，然后赢得干净利落！很快你就比我的三姨太更有钱了！

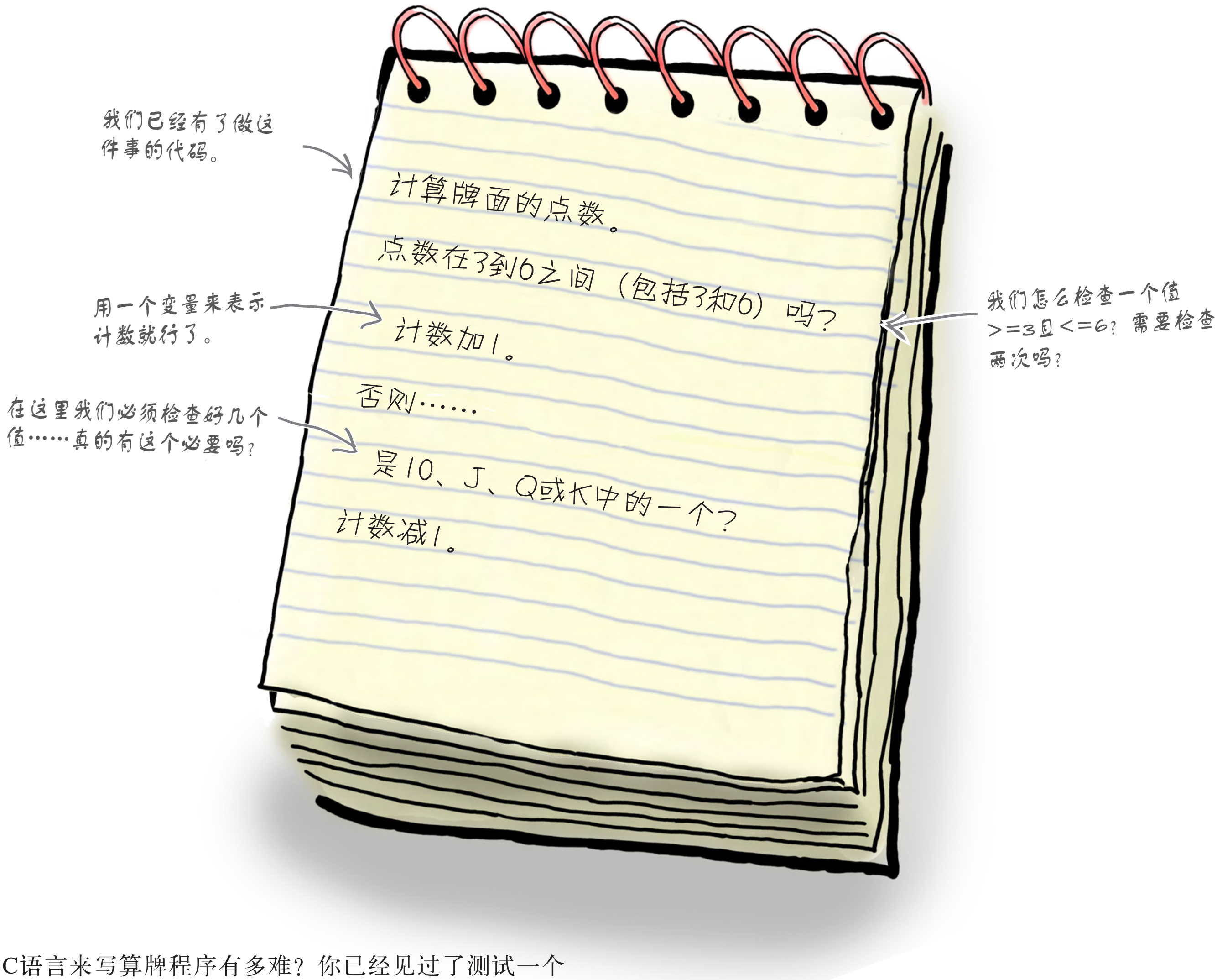
如果想学到更多的东西，今天就加入我的21点扑克技能专修学校，除了算牌，你还能学到：

* 如何利用“凯利公式”使赌注的价值最大化
* 如何与监赌人员周旋
* 如何去除真丝西服上的奶酪污渍
* 如何搭配花格子图案的服饰

详情请联系刀仔，由21点扑克技能专修学校转交。

用C语言算牌？

算牌是一种提高21点获胜几率的方法。在发牌时不断更新计数，玩家就可以计算出下大注和下小注的最佳时机。虽然这是一项强大的技术，但它真的非常简单。



用C语言来写算牌程序有多难？你已经见过了测试一个条件的方法，但算牌算法需要检查好几个条件：需要在检查一个数>=3的同时检查它<=6。

所以需要一组操作将多个条件组合在一起。

布尔运算

到目前为止，你已经见过了if语句，它检查一个条件是否为真。如果我们想要检查多个条件？或检查一个条件非真呢？

&&检查两个条件都为真

只有当给出的两个条件同时为真时，与运算（&&）的结果才为真。

```
if ((dealer_up_card == 6) && (hand == 11))
    double_down();
```

只有两个条件都为真，这段代码才会运行。

与运算的效率很高，因为如果第一个条件为假，计算机就不会自寻烦恼地去计算第二个条件，因为它知道如果第一个条件为假，那么整个条件也一定为假。

||检查两个条件中只要有一个为真

两个条件中只要有一个为真时，或运算（||）的结果就是真。

```
if (cupcakes_in_fridge || chips_on_table)
    eat_food();
```

只要有一个为真。

如果第一个条件为真，计算机就不会自找麻烦地去计算第二个条件，因为它知道只要第一个条件为真，整个条件也一定为真。

!把条件的值反过来

!是非运算，它将一个条件的值取反。

```
if (!brad_on_phone)
    answer_phone();
```

!表示“非”。

百宝箱

在C语言中，布尔值是用数字表示的。对C语言来讲，数字0代表假的值。那什么数字代表真呢？任何不等于0的数字都将被当成真处理，因此下面的C代码也没错：

```
int people_moshing = 34;
if (people_moshing)
    take_off_glasses();
```

事实上，C程序常用它作为“检查某个变量不为0”的简写。

为了让程序能用来算牌，请做一些修改。如果牌的点数在3到6之间，程序需要显示一条消息；如果牌是10、J、Q或K，则需要显示不同消息。

```
int main()
{
    char card_name[3];
    puts("输入牌名: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    /* 检查牌的点数是否在3到6之间 */
    if ..............................................................
        puts("计数增加");
    /* 否则，检查牌是否是10、J、Q或K */
    else if .........................................................
        puts("计数减少");
    return 0;
}
```

C标准礼貌指南

ANSI C标准没有用来表示真和假的值，C程序把0这个值当做假处理，把0以外的任何值当做真处理。C99标准则允许在程序中使用true和false关键字。但编译器还是会把它们当做1和0这两个值来处理。

练习解答

为了让程序能用来算牌，请做一些修改。如果牌的点数在3到6之间，程序需要显示一条消息；如果牌是10、J、Q或K，则需要显示不同消息。

```
int main()
{
    char card_name[3];
    puts("输入牌名: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    /* 检查牌的点数是否在3到6之间 */
    if ((val > 2) && (val < 7))
        puts("计数增加");
    /* 否则，检查牌是否是10、J、Q或K*/
    else if (val == 10)
        puts("计数减少");
    return 0;
}
```

这个条件有好几种写法。

这里只需要一个条件，你发现了吗？

这里没有蠢问题

问： 为什么不能只写一个|和&？

答： 也不是不行。&和|操作符总是计算两个条件，而&&和||可以跳过第二个条件。

问： 那还要|和&干什么呢？

答： 对逻辑表达式求值只是它们的一个用处，它们还能对数字的某一位进行布尔运算。

问： 那是什么意思？

答： 6 & 4等于 4，是因为当对6（二进制数110）和4（二进制数100）的每个二进制位布尔与时，就会得到4（二进制数100）。

试驾

现在编译并运行程序，看看会发生什么：

运行命令用来编译并运行代码。

我们多次运行程序，以检查程序在取不同值的情况下有不同的输出。

```
File Edit Window Help FiveOfSpades
> gcc cards.c -o cards && ./cards
输入牌名：
Q
计数减少

> ./cards
输入牌名：
8

> ./cards
输入牌名：
3
计数增加

>
```

代码正确运行。通过布尔运算符将多个条件组合在一起，就可以检查取值是否在某个范围内，而不仅仅是一个值。现在算牌器已经初具雏形。

编译器大曝光

本周访谈：gcc的奉献

Head First：gcc，非常谢谢您在百忙之中抽出时间接受我们的采访。

gcc：小事一桩，很高兴能参加你们的节目。

Head First：gcc，听说你会说很多种语言，是真的吗？

gcc：我熟练地掌握了600多万种沟通方式……

Head First：真的假的？

gcc：呵呵，开玩笑啦，不过我的确会说很多种语言，除了C语言，我还会C++和Objective-C，对Pascal、Fortran和PL/I等语言也有一定研究，Go语言我也略知一二……

Head First：在硬件方面，听说你可以生成很多平台的机器代码？

gcc：几乎任何处理器。一般而言，每当硬件工程师新创造了一种处理器，他要做的第一件事情就是让我在上面运行。

Head First：这种灵活性简直不可思议，请问你是怎么办到的呢？

gcc：我的秘诀就是拥有双重性格。我有一个前端，这个部分的我可以理解某种类型的源代码。

Head First：比如用C语言写的源代码？

gcc：没错，我的前端能够将这种语言转化为一种中间代码，所有的语言前端都能够生成同一种代码。

Head First：那么另外一种性格呢？

gcc：我还有一个后端，一个将中间代码转化为多种平台的机器代码的系统。每种操作系统都有自己特定的可执行文件格式，但我都知道……

Head First：可是人们通常仅仅将你描述为翻译器，你认为这公平吗？毕竟翻译不是你的全部。

gcc：是的，除了简单的翻译之外我还干很多事情，例如我会发现代码中的错误。

Head First：能举些例子吗？

gcc：我能够检查明显的错误，例如变量名拼错了；我也能找到不容易发现的错误，例如变量的重复定义；当程序员用已经存在的函数名去命名变量时，我也会发出警告，等等。

Head First：也就是说你会检查代码的质量？

gcc：没错，不仅仅是质量，还有性能。如果我发现循环中的某段代码提到循环外面执行时也一样正确，我会默默移动它。

Head First：你真的干了很多活！

gcc：是的，但我一向低调行事。

Head First：gcc，谢谢你接受我们的采访。

变身编译器

这页上的每个C文件都代表一个完整的源文件。你的工作是扮演编译器，并决定它们能否编译成功。如果不能，说明原因。如果你想得到附加分，说明程序编译以后的运行结果，以及它们是否能按预期工作。

A

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1)
        card = card - 1;
        if (card < 7)
            puts("小牌");
    else {
      puts("Ace!");
    }
    return 0;
}
```

B

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("小牌");
    else
      puts("Ace!");
    }
    return 0;
}
```

C

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("小牌");
    } else
      puts("Ace!");

    return 0;
}
```

D

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("小牌");
    else
      puts("Ace!");

    return 0;
}
```

变身编译器解答

这页上的每个C文件都代表一个完整的源文件。你的工作是扮演编译器，并决定它们能否编译成功。如果不能，说明原因。如果你想得到附加分，说明程序编译以后的运行结果，以及它们是否能按预期工作。

A

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1)
        card = card - 1;
        if (card < 7)
            puts("小牌");
    else {
      puts("Ace!");
    }
    return 0;
}
```

编译成功。程序显示“小牌”，但程序不能正确工作，因为else匹配了错误的if。

B

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("小牌");
    else
      puts("Ace!");
    }
    return 0;
}
```

编译成功。程序什么也没显示，它不能正确工作，因为else匹配了错误的if。

C

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("小牌");
    } else
      puts("Ace!");

    return 0;
}
```

编译成功。程序显示“Ace!”，正确！

D

```
#include <stdio.h>

int main()
{
    int card = 1;
    if (card > 1) {
        card = card - 1;
        if (card < 7)
            puts("小牌");
    else
      puts("Ace!");

    return 0;
}
```

编译失败。因为花括号不匹配。

现在的代码

```
int main()
{
    char card_name[3];
    puts("输入牌名: ");
    scanf("%2s", card_name);
    int val = 0;
    if (card_name[0] == 'K') {
        val = 10;
    } else if (card_name[0] == 'Q') {
        val = 10;
    } else if (card_name[0] == 'J') {
        val = 10;
    } else if (card_name[0] == 'A') {
        val = 11;
    } else {
        val = atoi(card_name);
    }
    /* 检查牌的点数是否在3到6之间 */
    if ((val > 2) && (val < 7))
        puts("计数增加");
    /* 否则，检查牌是否为10、J、Q或K */
    else if (val == 10)
        puts("计数减少");
    return 0;
}
```

C程序经常需要多次检查同一个值，并且在每一种情况中执行非常类似的代码片段。

可以使用一连串的if语句，这没有错，但对于这种逻辑，C语言提供了替代的写法。

C语言可以用switch语句进行逻辑测试。

随时转向的命运列车

有时候当你在写条件逻辑时，需要一次又一次地检查同一个变量的值。为了避免写许许多多的if语句，C语言提供了另一种选择：**switch**语句。

switch语句和if语句有些像，但它可以测试一个变量的多种取值：

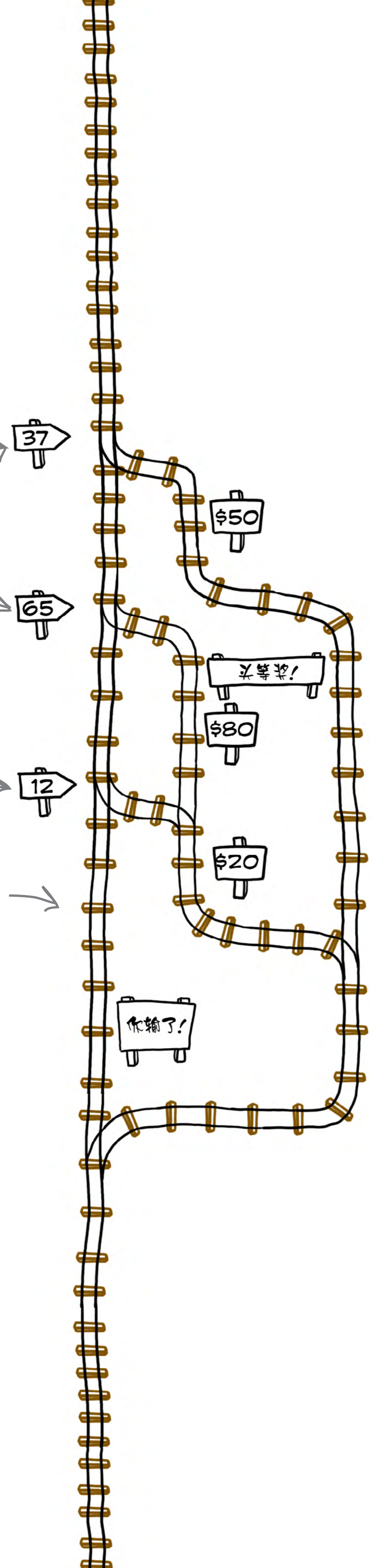
```
switch(train) {
case 37:
  winnings = winnings + 50;
  break;
case 65:
  puts("头等奖!");
  winnings = winnings + 80;
case 12:
  winnings = winnings + 20;
  break;
default:
  winnings = 0;
}
```

如果train == 37，winnings加50，然后跳到终点。

如果train == 65，winnings加80，再加20，然后跳到终点。

如果train == 12，winnings加20。

对于其他任意train的值，winnings归零。



当计算机遇到switch语句，它会检查给出的值，然后寻找匹配的case。找到后，它会运行case之后的所有代码直到遇到break语句。**计算机会一直运行下去直到有人吩咐它退出switch语句。**

漏掉break会让代码出错。

大部分C程序在每个case段的末尾都有一条break语句，这样做虽然会有失效率，但可以提高代码的可读性。

磨笔上阵

让我们再看一下cards程序中的那段代码：

```
int val = 0;
if (card_name[0] == 'K') {
    val = 10;
} else if (card_name[0] == 'Q') {
    val = 10;
} else if (card_name[0] == 'J') {
    val = 10;
} else if (card_name[0] == 'A') {
    val = 11;
} else {
    val = atoi(card_name);
}
```

你能用switch语句重写这段代码吗？把你的答案写在下面吧：

磨笔上阵 解答

请用switch语句重写代码。

```
int val = 0;
if (card_name[0] == 'K') {
    val = 10;
} else if (card_name[0] == 'Q') {
    val = 10;
} else if (card_name[0] == 'J') {
    val = 10;
} else if (card_name[0] == 'A') {
    val = 11;
} else {
    val = atoi(card_name);
}
```

```
int val = 0;
switch(card_name[0]) {
case 'K':
case 'Q':
case 'J':
    val = 10;
    break;
case 'A':
    val = 11;
    break;
default:
    val = atoi(card_name);
}
```

要点

- switch语句可以取代一连串的if语句。
- switch语句检查一个单独的值。
- 计算机会在第一个匹配的case语句处开始执行代码。
- 在遇到break或到达switch语句的末尾前，代码会一直运行。
- 核对是否把break放对了地方，否则switch语句就会出错。

这里没有蠢问题

问：为什么我要用switch语句取代if？

答： 当需要多次检查同一变量时，使用switch语句会更方便。

问：使用switch语句有什么好处？

答： 有这几个好处。第一，让代码更清晰，一段代码处理一个变量的结构，结构一目了然，相反，一连串的if语句就没那么清晰了；第二，可以用下落逻辑在不同的分支之间复用代码。

问：switch语句只能检查变量吗？它能检查值吗？

答： 能，switch语句仅仅检查两个值是否相等。

问：我能在switch语句中检查字符串吗？

答： 不能用switch语句检查字符串或任何形式的数组，switch语句只能检查值。

有时一次还不够……

你已经学习了很多关于C语言的知识，但仍有一些重要的东西需要了解。你已经知道了如何根据不同的情况写程序，但有一样基本的东西你还没见过，如果想让程序反复做一件事，怎么做？

在C语言中使用while循环

循环是一种特殊类型的控制语句。控制语句决定一段代码是否运行，而循环语句决定了一段代码运行几次。

C语言中最基本的循环结构是`while`循环，只要循环条件一直为真，`while`循环就会一次又一次、周而复始地运行代码。

在运行循环体前检查条件。

```
while (<某个条件>) {
    ... /* 在这里做一些事情 */
}
```

循环体在花括号里。

如果循环体中只有一行代码，你可以不加花括号。

当程序执行到循环体的末尾，计算机就会检查循环条件是否仍为真。如果是，就会再次执行循环体中的代码。

```
while (more_balls)
    keep_juggling();
```

> ### 今天你do while了吗？
>
> `while`循环还有一种形式，它总是在循环体运行后才检查循环的条件，也就是说循环体至少会被执行一次，我们叫它**do...while**循环：
>
> ```
> do {
>   /* 买彩票 */
> } while(have_not_won);
> ```

所有循环的结构都相同……

当重复执行一段代码时，可以用while循环。但循环的结构总是如出一辙。

- 在循环开始前做一些简单的工作，比如设置计数器。
- 在每一轮的循环开始前进行条件测试。
- 在每一轮的循环结束后做一些工作，如更新计数器。

下面这个例子是一个从1数到10的while循环：

```
int counter = 1;           ← 这是循环启动代码。
while (counter < 11) {     ← 这是循环条件。
  printf("%i个枣\n", counter);
  counter++;               ← 别忘了：counter++表示“把counter变量的值加1”。
}
```

这是循环更新代码，它用来在循环体的末尾更新计数器。

所有循环都是这样的三部曲：首先为循环准备变量，其次在每一轮的循环前检查条件，最后在循环末尾更新计数器或实现类似功能。

……for循环让事情变得更简单

因为这个模式是通用的，C语言的设计者就创造了**for**循环，它让代码看起来更简洁。同样的代码如果用for循环来写：

```
int counter;
for (counter = 1; counter < 11; counter++) {
  printf("%i个枣\n", counter);
}
```

- 初始化循环变量。（counter = 1）
- 这是每次循环执行前对条件进行检查的代码。（counter < 11）
- 这是每次循环后运行的代码。（counter++）
- 因为循环体中只有一行代码，可以去掉花括号。

C程序大量使用for循环，至少要和while循环用得一样多。for循环不但可以减少代码的行数，而且便于其他C程序员阅读，因为所有用来控制循环的代码——控制counter变量值的代码——现在都放到了for语句中，并从循环体中分离了出来。

每个for循环的循环体里都需要有点东西。

用break语句退出循环……

但如果想在循环中的某个地方跳出循环呢？当然，可以重新调整代码的结构，但更简单的方法是，使用**break**语句直接跳出循环：

```
while (feeling_hungry) {
    eat_cake();
    if (feeling_queasy) {
        /* 从while循环中跳出  */
        break;
    }
    drink_coffee();
}
```

"break"直接跳出循环。

break语句可以直接退出当前循环，跳过循环体中break之后的所有语句。break非常有用，因为它有时是结束循环最简单有效的方法，但应该避免滥用break，因为它们会降低代码的可读性。

break语句可以用来退出循环语句和switch语句。

使用break时看清你在哪里，并不是所有地方都能够使用break。

……用continue继续循环

如果想跳过循环体的其余部分，然后回到循环的开始，那么continue语句就是你的最佳伴侣：

```
while (feeling_hungry) {
    if (not_lunch_yet) {
        /* 回到循环条件  */
        continue;
    }
    eat_cake();
}
```

"continue"带你回到循环的开始。

古墓谜案

break不能从if语句中退出。

1990年1月15日，AT&T的长途电话系统死机，造成6万人无法使用电话服务。起因是一个负责写电路交换部分C代码的开发人员企图用break从if语句中退出，但break不能从if语句中退出。相反，程序跳过了整段代码，引起了这个bug，令7千万次电话呼叫在9个多小时内无法接通……

函数聚焦

在试验新学的循环“咒语”前，我们绕道去看一眼函数。

到目前为止，在你写过的每个程序中，都必须创建一个函数——main()函数：

```
int main()
{
    puts("人生须臾，芳华易逝");
    return 0;
}
```

这个函数返回一个整型值。

这是函数的名字。

这对括号中什么都没有。

函数体——做事情的那部分。

函数体被花括号包围。

事情做完以后，返回一个值。

在C语言中几乎所有函数都有着相同的格式。例如在下面这个程序中，main()函数调用了一个自定义函数。

```
#include <stdio.h>

int larger(int a, int b)
{
  if (a > b)
    return a;
  return b;
}

int main()
{
  int greatest = larger(100, 1000);
  printf("%i最大!\n", greatest);
  return 0;
}
```

返回整型值

这个函数接收两个参数：a和b。这两个参数都是整型。

在这里调用函数

larger()函数与main()函数有一点区别，它接收参数（argument或parameter）。参数是一个局部变量，函数从调用它的代码那里得到参数的值。larger()函数要接收两个参数：a和b，它返回两个参数中较大那个的值。

C标准礼貌指南

main()这一函数的返回类型是int，因此必须在函数结束前包含一条return语句。即使不加，代码也能编译通过，但会收到编译器的警告。支持C99标准的编译器会在你忘记的时候插入一条return语句。如果你想让编译器遵循C99标准，可以使用**-std=99**选项。

void函数聚焦

在C语言中，大部分函数都有一个返回值。但有时候，想要创建的函数中并没有有用的信息需要返回。它更多只是做了一些事，而不是计算出一个结果。通常情况下，函数都需要包含一条return语句，但只要把函数的返回类型声明为void，没有return语句也无妨。

返回类型为void表示这个函数不会返回任何东西。

```
void complain()
{
    puts("我真的不快乐");
}
```

因为是void函数，所以没有必要写return语句。

在C语言中，关键字void意味着无所谓，一旦告诉C编译器你不关心函数的返回值，函数就不需要有return语句。

这里没有蠢问题

问： 如果我创建了一个void函数，是否就意味它一定不能有return语句？

答： 你还是可以包含return语句，但编译器很可能会产生一条警告消息。而且在void函数中包含return语句没有任何意义。[①]

问： 真的吗？为什么没有意义？

答： 因为如果你试图读取void函数的值，编译器会报错。

链式赋值

在C语言中，几乎每样东西都有返回值，不仅仅是函数调用，就连赋值表达式也有返回值。例如下面这条语句：

```
x = 4;
```

它把数字4赋值给变量。有趣的是表达式“x = 4”本身也有一个值，这个值是4，即赋给x的值。为什么说这个东西很有用呢？因为你可以用它来做一些很酷的事情，比如把多条赋值语句链在一起写：

```
y = (x = 4);
```

赋值表达式“x = 4”的值是4。

所以现在y也设为了4。

这行代码同时将x和y的值设为了4。事实上，可以去掉括号，缩短代码的长度：

```
y = x = 4;
```

你经常会在需要给多个变量赋相同值的代码中看到链式赋值。

① 在void函数中的return语句有时可以用来提前退出函数。——译者注

弄乱的消息

下面列出了一个C语言的小程序。程序少了一块代码，你的任务是将候选代码块（左）和它对应的输出结果进行配对。有的输出结果可能一次都用不到，有的可能用到好几次。用直线把候选代码块和它所对应的命令行输出连接起来。

```
#include <stdio.h>

int main()
{
    int x = 0;
    int y = 0;
    while (x < 5) {
        [                    ]
        printf("%i%i ", x, y);
        x = x + 1;
    }
    return 0;
}
```

候选代码放在这里。

候选项：

```
y = x - y;
```

```
y = y + x;
```

将每一个候选项和可能的输出结果配对。

```
y = y + 2;
if (y > 4)
    y = y - 1;
```

```
x = x + 1;
y = y + x;
```

```
if (y < 5) {
    x = x + 1;
    if (y < 3)
        x = x - 1;
}
y = y + 2;
```

可能的输出结果：

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

练习

既然你已经知道了怎么创建while循环，请修改程序让它在游戏期间保持计数。每发一张牌就显示一次计数，如果玩家输入X就终止程序，如果玩家输入了错误的值（如11或24）就显示错误消息。

```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    int count = 0;
    do {
        puts("输入牌名： ");
        scanf("%2s", card_name);
        int val = 0;
        switch(card_name[0]) {
        case 'K':
        case 'Q':
        case 'J':
            val = 10;
            break;
        case 'A':
            val = 11;
            break;
        case 'X':        ← 你将在这里做什么？
            ..........................................
        default:
            val = atoi(card_name);
            ..........................................   ← 如果val不在1到10之间，就显示一条错误消息。你还应该跳过循环体的其余部分，然后再试一次。
            ..........................................
            ..........................................
            ..........................................
            ..........................................
        }
        if ((val > 2) && (val < 7)) {
            count++;          ← 计数加1。
        } else if (val == 10) {
            count--;          ← 计数减1。
        }
        printf("当前的计数： %i\n", count);
    } while (..............);   ← 如果用户输入了X，就停止程序。
    return 0;
}
```

弄乱的消息解答

下面列出了一个C语言的小程序。程序少了一块代码，你的任务是将候选代码块（左）和它对应的输出结果进行配对。有的输出结果可能一次都用不到，有的可能用到好几次。请用直线把候选代码块和它所对应的命令行输出连接起来。

```
#include <stdio.h>

int main()
{
    int x = 0;
    int y = 0;
    while (x < 5) {

        printf("%i%i ", x, y);
        x = x + 1;
    }
    return 0;
}
```

候选代码放在这里。

候选项：

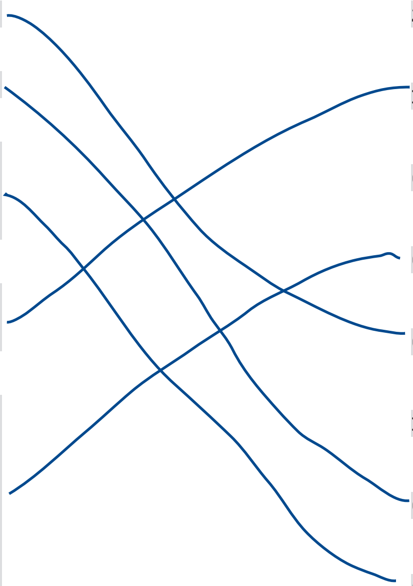
```
y = x - y;
```

```
y = y + x;
```

```
y = y + 2;
if (y > 4)
    y = y - 1;
```

```
x = x + 1;
y = y + x;
```

```
if (y < 5) {
    x = x + 1;
    if (y < 3)
        x = x - 1;
}
y = y + 2;
```



可能的输出结果：

```
22 46
```

```
11 34 59
```

```
02 14 26 38
```

```
02 14 36 48
```

```
00 11 21 32 42
```

```
11 21 32 42 53
```

```
00 11 23 36 410
```

```
02 14 25 36 47
```

练习解答

既然你已经知道了怎么创建while循环，请修改程序让它在游戏期间保持计数。每发一张牌就显示一次计数，如果玩家输入X就终止程序，如果玩家输入了错误的值（如11或24）就显示错误消息。

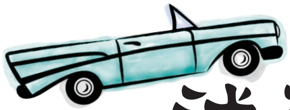
```
#include <stdio.h>
#include <stdlib.h>
int main()
{
    char card_name[3];
    int count = 0;
    do {
        puts("输入牌名： ");
        scanf("%2s", card_name);
        int val = 0;
        switch(card_name[0]) {
        case 'K':
        case 'Q':
        case 'J':
            val = 10;
            break;
        case 'A':
            val = 11;
            break;
        case 'X':
            continue;
        default:
            val = atoi(card_name);
            if ((val < 1) || (val > 10)) {
                puts("我无法理解这个值！");
                continue;
            }
        }
        if ((val > 2) && (val < 7)) {
            count++;
        } else if (val == 10) {
            count--;
        }
        printf("当前的计数： %i\n", count);
    } while ( card_name[0] != 'X' );
    return 0;
}
```

在这里用break不能退出循环，因为我们现在位于switch语句中。我们需要用continue回到循环开始，然后再次检查循环条件。

这只是这个条件的一种写法。

在这个地方还需要一个continue，因为你想让循环继续下去。

需要检查第一个字符是否是X。



试驾

既然算牌程序已经完成了，是时候带它出去兜兜风了，您意下如何？觉得它能工作吗？

别忘了：如果在Windows中，就不需要加“./”。

这条命令将编译并运行程序。

现在我们就要检查一下输入是否正确了。

计数在增加！

```
File Edit Window Help GoneLoopy
> gcc card_counter.c -o card_counter && ./card_counter
输入牌名：
4
当前的计数： 1
输入牌名：
K
当前的计数： 0
输入牌名：
3
当前的计数： 1
输入牌名：
5
当前的计数： 2
输入牌名：
23
我无法理解这个值！
输入牌名：
6
当前的计数： 3
输入牌名：
5
当前的计数： 4
输入牌名：
3
当前的计数： 5
输入牌名：
X
```

算牌程序工作了！

你已经完成了第一个C程序。借助C语言的语句、循环、条件的威力，你已经创造了一个具有完整功能的算牌器。

干得好！

免责声明：用计算机算牌在很多州是犯法的，赌场那群家伙可不是好惹的。所以千万别那么做，好吗？

这里没有蠢问题

问： C语言为什么需要编译？其他一些语言就不需要编译，比如JavaScript，是吗？

答： 为了让代码执行起来更快，C语言需要编译。尽管有些语言不是编译型语言，但它们中的一些，像JavaScript和Python，为了提高速度通常会在幕后使用一些编译技术。

问： C++是另一个版本的C语言吗？

答： 不是，虽然C++的设计初衷是为了扩展C，但现在看来远不止如此，人们最初创造C++和Objective-C都是为了用C语言写面向对象的程序。

问： 什么是面向对象？我们在本书中会学吗？

答： 面向对象是一种对抗软件复杂性的技术，我们在本书中不会做专门研究。

问： C语言为什么看起来很像JavaScript、Java和C#等语言？

答： C语言的语法非常简洁，因此影响了很多其他语言。

问： gcc这三个字母分别代表什么含义？

答： GNU编译器套装（GNU Compiler Collection）。

问： 为什么是“套装”？难道不止C语言一种吗？

答： GNU编译器套装可以用来编译很多语言，而C语言可能是人们在应用gcc时使用最多的语言。

问： 我能创建一个永无止尽的循环吗？

答： 可以，如果循环条件的值是1，循环就会永无止尽地运行下去。

问： 创建一个永无止尽的循环是个好主意吗？

答： 有时候是，通常在一些诸如网络服务器的程序中会使用无限循环（一个永无止尽的循环），程序会反复地做一件事直到有人停止它。但大部分的程序员使用循环是为了让它们在某个时刻停止。

要点

- 只要条件为真，while循环就会运行代码。
- do-while循环和while循环十分类似，不过至少执行一次代码。
- 某些循环用for来写更简洁。
- 可以用break在任意时刻退出循环。
- 可以用continue随时跳到循环条件处。
- return语句会从函数返回一个值。
- void函数不需要return语句。
- 在C语言中，所有表达式都有值。
- 赋值表达式有一个值，因此可以把它们链在一起写（x = y = 0）。

C语言工具箱

你已经学完了第1章，C语言的基础知识现在已经加入你的工具箱中了。关于本书的提示工具条的完整列表，请见附录ii。

switch语句高效地检查了一个变量的多种取值。

简单的语句就是命令。

块语句被{和}包围。

每个程序都需要一个main()函数。

需要在运行之前先编译C程序。

#include将外部代码（如用来输入输出的代码）包含进来。

如果条件为真，if语句就会运行代码。

可以在命令行中用&&操作符在编译之后马上运行程序，前提是必须编译成功。

-o指定了输出文件。

可以用&&和||把多个条件组合在一起。

gcc是最流行的C编译器。

源文件的文件名应该以.c结尾。

只要条件为真，while就会重复执行代码。

do-while至少执行一次代码。

count++表示计数加1。

count--表示计数减1。

用for写循环更简洁。

2 存储器和指针

指向何方？

如果真的想玩转C语言，就需要理解C语言如何操纵存储器。

C语言在如何使用存储器方面赋予了你更多的掌控权。在本章中，你将揭开存储器神秘的面纱，看到读写变量时到底发生了什么；学习数组的工作原理，以及怎样避免烦人的存储器错误；最重要的是，你将看到掌握指针和存储器寻址对成为一名地道的C程序员来讲有多么重要。

C代码包含指针

指针是理解C语言最基本的要素之一。那么什么是指针？指针就是存储器中某条数据的地址。

之所以要在C语言中使用指针有以下几个原因。

为了更好地理解指针，请放慢阅读的脚步！

1. **在函数调用时，可以只传递一个指针，而不用传递整份数据。**

2. **让两段代码处理同一条数据，而不是处理两份独立的副本。**

指针做了两件事：避免副本和共享数据。但既然指针只是地址而已，为什么它会令很多人感到困惑呢？因为指针是一种间接形式的地址。在茫茫存储器中追逐指针，一不小心就会迷路。而学习C指针的诀窍就是慢慢来。

轻松一刻

不要一口气看完本章。

虽然指针是个简单的概念，但要完全理解指针需要花时间慢慢来。休息一下，多喝点水，如果被它难住了，干脆泡个舒服的热水澡！

深入挖掘存储器

为了理解什么是指针，需要切开计算机的存储器瞧瞧。

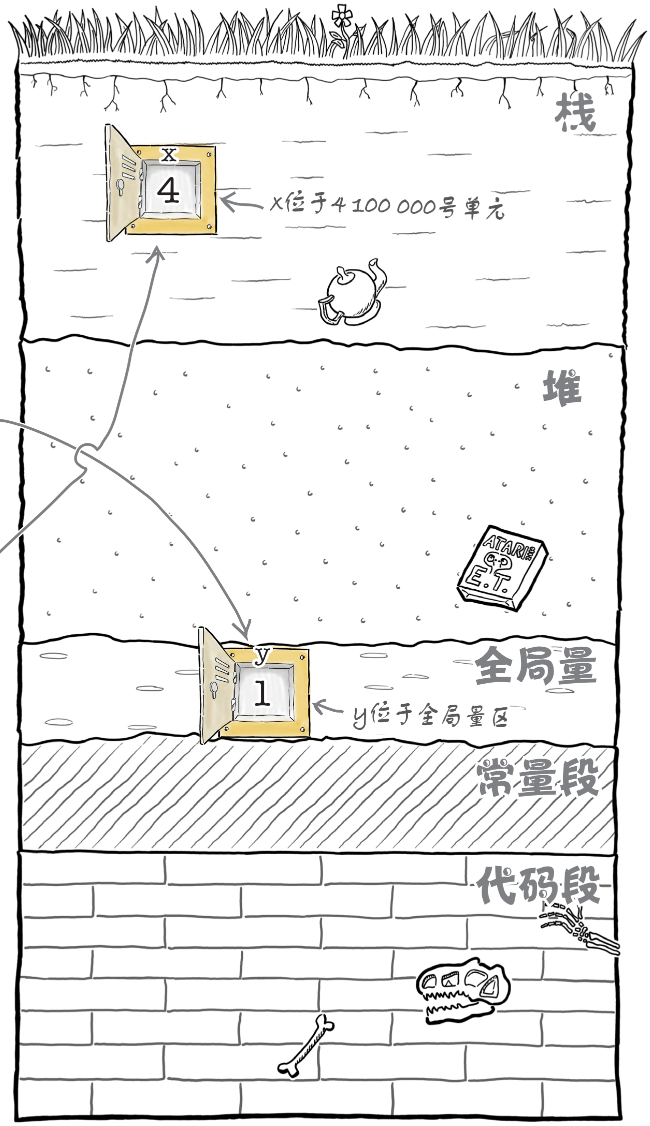
每当声明一个变量，计算机都会在存储器中某个地方为它创建空间。如果在函数（例如main()函数）中声明变量，计算机会把它保存在一个叫栈（Stack）的存储器区段中；如果你在函数以外的地方声明变量，计算机则会把它保存在存储器的全局量段（Globals）。

```
int y = 1;
```

变量y位于全局量段。存储器地址1 000 000。值为1。

```
int main()
{
  int x = 4;
  return 0;
}
```

变量x位于栈中，存储器地址4 100 000。值为4。



比如说，计算机可能将栈中4 100 000号存储器单元分配给变量x。如果把4赋给变量x，计算机就会把4保存在4 100 000号单元。

如果想要找出变量的存储器地址，可以用&运算符：

&x是x的地址。

```
printf("x保存在 %p\n", &x);
```

%p用来格式化地址。

这是代码打印的结果。

```
x保存在 0x3E8FA0
```

这是4 100 000的十六进制（以16为基数）表示。

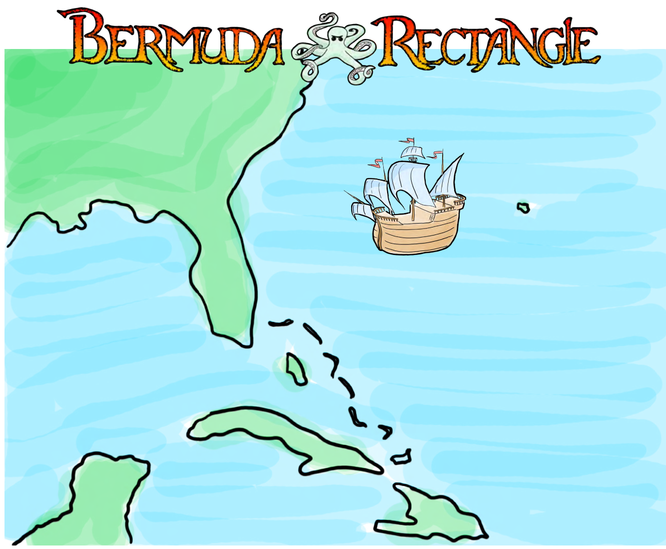
在你的机器上可能得到不同的地址。

在函数中声明的变量通常保存在栈中。
在函数外声明的变量保存在全局量区。

变量的地址告诉你去哪里找存储器中的变量，这就是为什么地址有时也叫指针，因为它指向了存储器中的变量。

和指针起航

想象你在为一个游戏编写程序，游戏中玩家需要控制船的航向……



游戏需要控制很多东西，比如得分、生命值和玩家当前的位置。你不想把游戏写成一段很长的代码，而是可以创建许多小的函数，让每个函数完成游戏的一个功能。

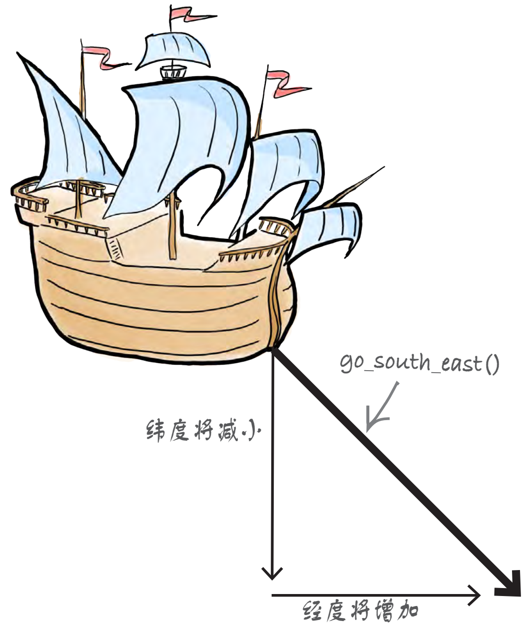
`go_south_east()`　`go_north_west()`　`speaks_in_present_tense()`

`go_south()`　`acquire_facial_hair()`

`eat_rat()`　`die_of_scurvy()`　`make_one_sequel_too_many()`

那么这和指针有什么关系？让我们先不考虑指针，写写看。你将和往常一样使用变量，游戏的主要部分是驾驶你的船在百慕大三角航行，我们具体看看代码需要在航行函数中完成哪些事。

船长，向东航行！



游戏用纬度（latitude）和经度（longitude）记录玩家的位置，纬度标记玩家南北方向的位置，经度标记玩家东西方向的位置。如果玩家想要向东南方向航行，他的纬度将减小，经度将增加：

于是可以写一个`go_south_east()`函数，它接收`latitude`和`longitude`这两个变量，然后对它们进行加、减操作：

```
#include <stdio.h>

void go_south_east(int lat, int lon)
{
  lat = lat - 1;
  lon = lon + 1;
}

int main()
{
  int latitude = 32;
  int longitude = -64;
  go_south_east(latitude, longitude);
  printf("停！当前位置：[%i, %i]\n", latitude, longitude);
  return 0;
}
```

传入latitude和longitude

纬度减小

经度增加

程序开始时船的位置是[32, −64]，如果它向东南方向航行，船的新坐标将是[31, −63]，前提是代码正确工作……

脑力风暴

仔细看看这段代码，你认为它能正确工作吗？为什么？

试驾

程序应该将船从[32, －64]向东南方向移动到[31, －63]，但编译并运行程序，结果却是：

怎么回事？船还停在老地方。

我们的力气都白花了吗？

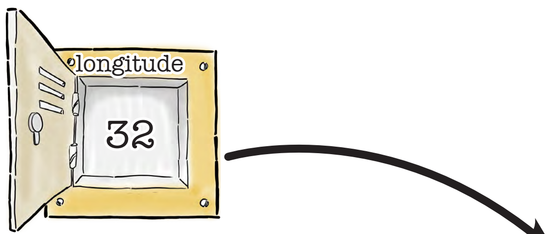
```
File Edit Window Help Savvy?
> gcc southeast.c -o southeast
> ./southeast
停！当前位置： [32, -64]
>
```

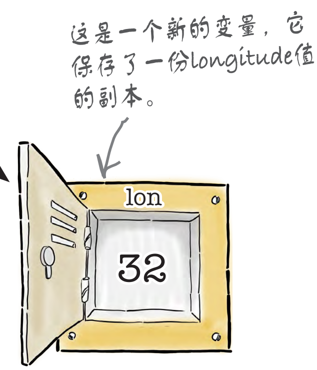
船准确地停在了原来的位置。

C语言按值传递参数

C语言调用函数的方式是导致这段代码不能正确工作的原因。

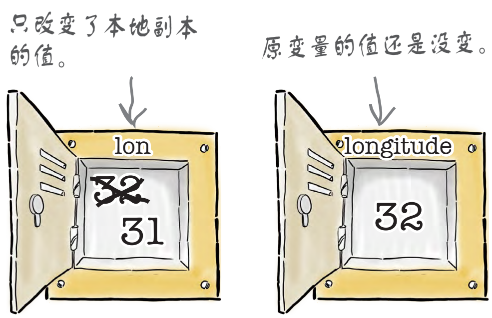
1. 一开始，main()函数有一个叫longitude的局部变量，它的值是32。





2. 当计算机调用go_south_east()函数时，它将变量longitude的值复制给了参数lon，这只是一个赋值的过程，从变量longitude到变量lon。也就是说，当你调用函数时，作为参数传递的不是变量，而只是变量的值。

3. 当go_south_east()函数修改了lon的值时，函数只是修改了本地的副本，也就是说程序返回main()函数时，变量longitude中保存的还是它原来的值32。

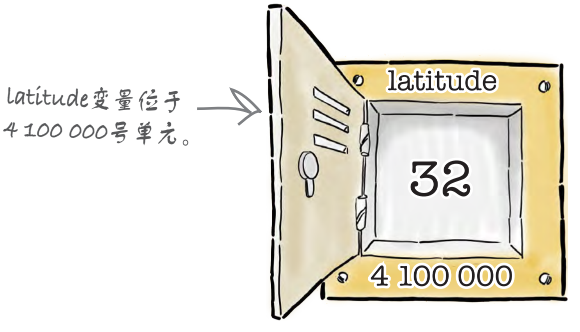


既然C语言就是这样调用函数的，那么怎么才能在函数中更新变量呢？

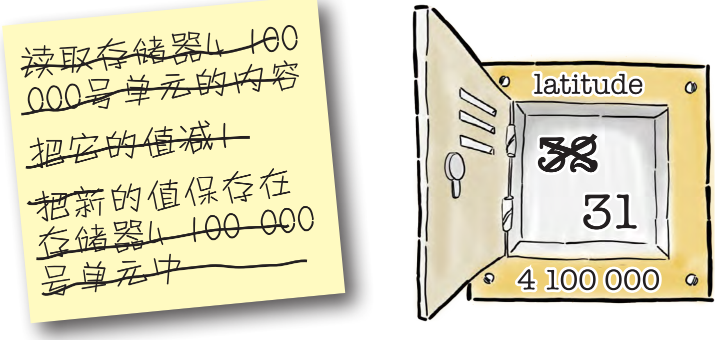
如果用指针，事情就好办多了……

试着传递指向变量的指针

如果传递的是变量latitude和longitude的地址，而不是它们的值，会怎么样？如果变量longitude位于存储器栈4 100 000号单元，当把4 100 000这个单元号作为参数传递给go_south_east()函数会发生什么？



如果告诉go_south_east()函数latitude的值位于4 100 000号单元，它不仅能找到latitude当前的值，而且还能够修改原latitude变量中的内容。函数所需要做的就是读取和更新存储器4 100 000号单元的内容。



因为go_south_east()函数更新了原latitude变量的值，计算机就能在返回main()函数后打印出更新后的坐标。

指针让存储器易于共享

使用指针的主要原因之一就是让函数共享存储器。一个函数可以修改另一个函数创建的数据，只要它知道数据在存储器中的位置。

既然你知道了使用指针修复go_south_east()函数的理论，是时候看看如何操作了。

这里没有蠢问题

问： 我在自己的机器上打印出了变量的单元号，但它不是4 100 000。是不是哪里做错了？

答： 你没有做错，在不同机器中，程序用来保存变量的存储器单元号不同。

问： 为什么局部变量保存在栈里，而全局变量保存在其他地方？

答： 局部变量和全局变量的用法不同。你永远只能得到一份全局变量，但如果写了一个调用自己的函数，就会得到同一个局部变量的很多个实例。

问： 存储器中的其他区域是用来做什么的？

答： 你会在本书的后续章节中看到它们的作用。

使用存储器指针

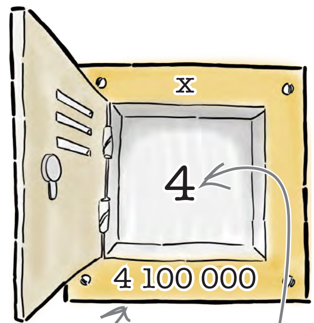
为了使用指针读写数据，需要了解三件事。

1 得到变量的地址。

可以用&运算符找到变量保存在存储器中的位置，这你已经知道了：

%p格式符将地址以16进制（以16为基数）的格式输出。

```
int x = 4;
printf("x lives at %p\n", &x);
```



&运算符将找到变量的地址：4 100 000。

一旦得到了变量的地址，就需要把它保存在某个地方。为此，需要指针变量。指针变量是一个用来保存存储器地址的变量。当声明指针变量时，需要说明指针所指向的地址中保存的数据的类型：

这是一个指针变量，它保存的是一个地址，这个地址中保存的是一个int型变量。

```
int *address_of_x = &x;
```

2 读取地址中的内容。

当你有了存储器地址，就想读取保存在那里的数据，这时可以用*运算符：

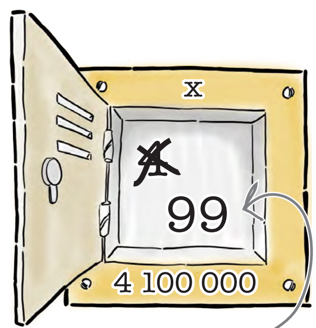
```
int value_stored = *address_of_x;
```

它会读取address_of_x所给出的存储器地址中的内容。它将会被设置为4：这个值是一开始就保存在变量x中的值。

*运算符和&运算符恰好相反。&运算符接收一个数据，然后告诉你这个数据保存在哪里；*运算符接收一个地址，然后告诉你这个地址中保存的是什么数据。因为指针有时也叫引用，所以*运算符也可以描述成对指针进行解引用。

3 改变地址中的内容。

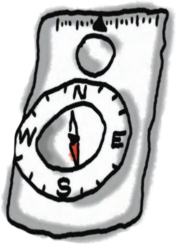
如果你有一个指针变量，并想修改这个变量所指向地址中的数据，可以再次使用*运算符，只不过这次需要把指针变量放在赋值运算符的左边。




```
*address_of_x = 99;
```

它会把原x变量中的内容改成99。

既然你知道了如何读写某个存储器单元的内容，下面就来修复go_south_east()函数吧。



指南针冰箱贴

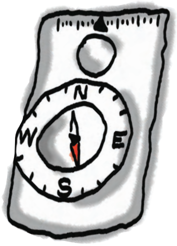
现在你需要修复go_south_east()函数，让它用指针更新正确的数据。仔细地想一下需要传给函数什么类型的数据，更新船的位置时应该使用什么运算符。

什么类型的参数可以保存int型变量的地址？

```
#include <stdio.h>

void go_south_east(.............. lat, .............. lon)
{
  .............. = .............. - 1;
  .............. = .............. + 1;
}

int main()
{
  int latitude = 32;
  int longitude = -64;

  go_south_east(..............,  ..............);
  printf("停! 当前位置: [%i, %i]\n", latitude, longitude);
  return 0;
}
```

别忘了：你将传递变量的地址。

&lon
*latitude
int
*lat
*lat
&lat
*lon
*lon
&lon
*longitude
&longitude
int *
lat
&latitude
int
&lat
lon
int *



指南针冰箱贴解答

现在你需要修复go_south_east()函数，让它用指针更新正确的数据。仔细地想一下需要传给函数什么类型的数据，更新船的位置时应该使用什么运算符。

参数将保存指针变量，因此它们必须是int *类型。

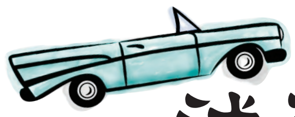
```
#include <stdio.h>

void go_south_east(int * lat, int * lon)
{
    *lat = *lat - 1;
    *lon = *lon + 1;
}

int main()
{
  int latitude = 32;
  int longitude = -64;

  go_south_east(&latitude, &longitude);
  printf("停! 当前位置: [%i, %i]\n", latitude, longitude);
  return 0;
}
```

*lat可以读取旧的值，并设置新的值。

需要用&运算符来找到latitude和longitude变量在存储器中的地址。

&lon　*latitude　int　&lat　&lon　*longitude

lat　int　&lat　lon

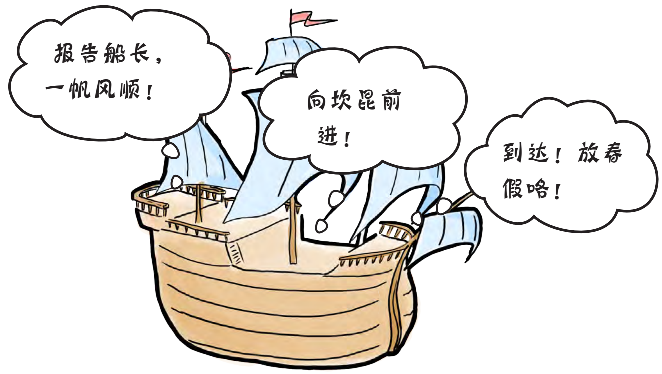


试驾

编译并运行新函数，你将得到：

原位置的东南方向。

```
File Edit Window Help Savvy?
> gcc southeast.c -o southeast
> ./southeast
停! 当前位置: [31, -63]
>
```



代码正确运行了。

函数接收指针作为参数，因此能够更新原latitude和longitude变量。现在不但能让函数返回某个值，还能让它更新某个存储器单元，只要把单元的地址传给它。

要点

- 计算机会为变量在存储器中分配空间。
- 局部变量位于栈中。
- 全局变量位于全局量段。
- 指针只是一个保存存储器地址的变量。
- &运算符可以找到变量的地址。
- *运算符可以读取存储器地址中的内容。
- *运算符还可以设置存储器地址中的内容。

这里没有蠢问题

问：指针是真实的地址单元，还是某种形式的引用？

答： 它们是进程存储器中真实编号的地址。

问：为什么存储器是进程的？

答： 计算机会为每个进程分配一个简版存储器，看起来就像是一长串字节。

问：但存储器并非如此？

答： 实际的存储器复杂多了，但细节对进程隐藏了起来，这样操作系统就可以在存储器中移动进程，或释放并重新加载到其他位置。

问：存储器不仅仅是一长列字节？

答： 物理存储器的结构十分复杂，计算机通常会将存储器地址分组映射到存储芯片的不同的存储体（memory bank）。

问：我需要理解它是怎么映射的吗？

答： 对大部分程序来说，你不需要关心机器组织存储器的细节。

问：为什么我一定要用`%p`格式串来打印指针？

答： 不一定要用`%p`，在大多数的现代计算机上可以用`%li`，但编译器可能会给出一条警告。

问：为什么`%p`以十六进制显示存储器地址？

答： 工程师通常以十六进制表示存储器地址。

问：如果我们把读取存储器单元的内容称为“解引用”，那么指针是不是应该叫“引用”？

答： 人们有时会把指针叫做“引用”，因为它引用了存储器中的某个地址单元。但C++程序员通常用“引用”表示C++中一个稍有不同的概念。

问：太好了，C++，我们会学到吗？

答： 别想了，这本书只教C语言。

怎么把字符串传给函数？

你知道了怎么把一些简单的值以参数的形式传给函数，但如果想传一些更复杂的东西呢？比如字符串。如果你还记得上一章的内容，就知道在C语言中字符串其实是字符数组，也就是说如果你想把字符串传给函数，可以这么做：

```
void fortune_cookie(char msg[])
{
  printf("Message reads: %s\n", msg);
}

char quote[] = "Cookies make you fat";
fortune_cookie(quote);
```

传一个字符数组给函数。

你把参数msg定义为数组，但是因为不知道字符串有多长，所以msg没有长度。这似乎很简单，但奇怪的事发生了……

亲爱的，谁截了我们的字符串？

C语言中有一个叫**sizeof**的运算符，它能告知某样东西在存储器中占多少字节，既可以对数据类型使用，也可以对某条数据使用。

在大多数计算机中，将返回4这个值。

```
sizeof(int)
sizeof("Turtles!")
```

将返回9，其中包含8个字符外加\0结束字符。

但当你检查传入函数字符串的长度时，离奇的事情发生了：

```
void fortune_cookie(char msg[])
{
  printf("Message reads: %s\n", msg);
  printf("msg occupies %i bytes\n", sizeof(msg));
}
```

为什么是8？在一些计算机中，这个数字甚至是4！为什么？

```
File Edit Window Help TakeAByte
> ./fortune_cookie
Message reads: Cookies make you fat
msg occupies 8 bytes
>
```

程序并没有显示字符串总长，而是返回了4或8个字节。发生了什么事？为什么fortune_cookie()认为我们传进来的字符串比实际要短？

脑力风暴

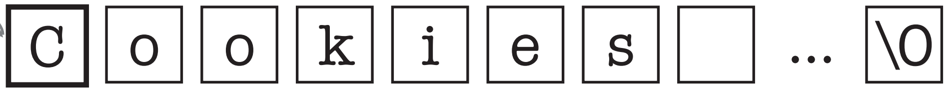
你认为是什么原因导致sizeof(msg)小于整个字符串的长度？msg是什么？为什么在不同的计算机上返回的大小不同？

数组变量好比指针……

当你创建了一个数组，数组变量就可以当作**指针**使用，它指向数组在存储器中的起始地址。当C语言在函数中看到这样一行代码时：

```
char quote[] = "Cookies make you fat";
```

quote变量代表字符串中第一个字符的地址。



计算机会为字符串的每一个字符以及结束字符\0在栈上分配空间，并把**首字符的地址**和quote变量关联起来，代码中只要出现这个quote变量，计算机就会把它替换成字符串首字符的地址。其实，数组变量就好比一个指针：

quote虽然是数组，但可以当指针变量来用。

```
printf("The quote 字符串保存在: %p\n", quote);
```

如果你写个测试程序来显示数组的地址，就会看到这样的结果。

```
File Edit Window Help TakeAByte
> ./where_is_quote
The quote 字符串保存在: 0x7fff69d4bdd7
>
```

……所以传给函数的是指针

这就是为什么fortune_cookie()代码发生了奇怪的事情。看起来把字符串传给了fortune_cookie()函数，但实际上只传了一个指向字符串的指针：

msg其实是指针变量。

msg指向传进来的消息。

```
void fortune_cookie(char msg[])
{
  printf("Message reads: %s\n", msg);
  printf("msg occupies %i bytes\n", sizeof(msg));
}
```

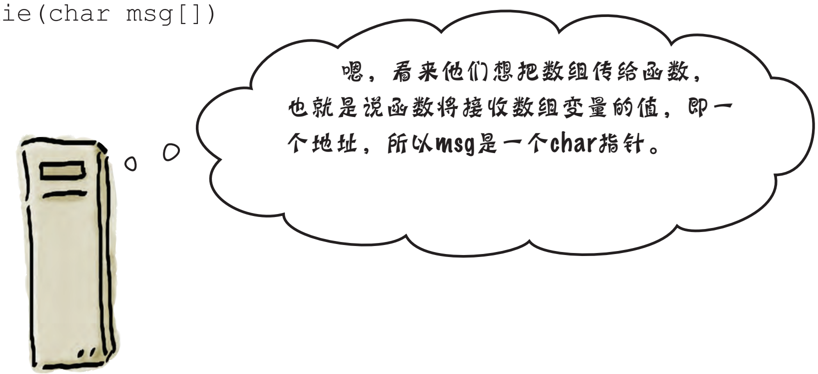
sizeof(msg)不过是指针变量的大小罢了。

这就是为什么sizeof运算符会返回奇怪结果，它只是返回了字符串指针的大小。指针在32位操作系统中占4字节，在64位操作系统中占8字节。

运行代码时，计算机在想什么

1 计算机看到函数。

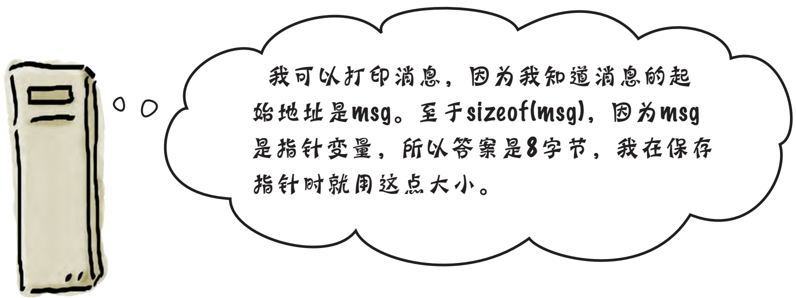
```
void fortune_cookie(char msg[])
{
  ...
}
```



2 紧接着，计算机看到了函数的内容。

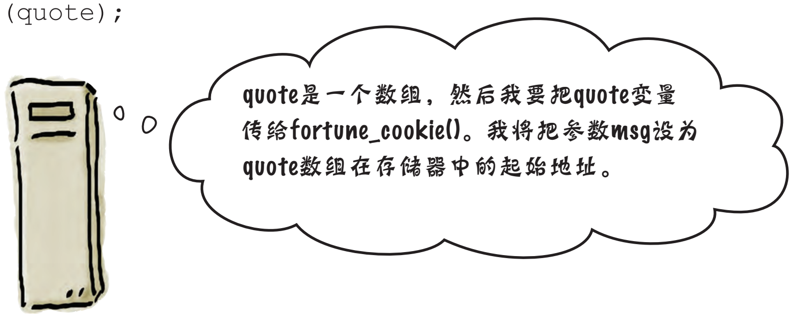
```
printf("Message reads: %s\n", msg);
printf("msg occupies %i bytes\n", sizeof(msg));
```

邮电



3 计算机调用函数。

```
char quote[] = "Cookies make you fat";
fortune_cookie(quote);
```



要点

- 数组变量可以被用作指针。
- 数组变量指向数组中第一个元素。
- 如果把函数参数声明为数组，它会被当作指针处理。
- sizeof运算符返回某条数据占用空间的大小。
- 也可以对某种数据类型使用sizeof，例如sizeof(int)。
- sizeof(指针)在32位操作系统中返回4，在64位操作系统中返回8。

这里没有蠢问题

问： **sizeof是一个函数吗？**

答： 不是，它是一个运算符。

问： **有什么区别？**

答： 编译器会把运算符编译为一串指令；而当程序调用函数时，会跳到一段独立的代码中执行。

问： **所以程序是在编译期间计算sizeof的？**

答： 没错，编译器可以在编译时确定存储空间的大小。

问： **为什么在不同的计算机上指针变量的大小不同？**

答： 在32位操作系统中，存储器地址以32位数字的形式保存，所以它叫32位操作系统。32位==4字节，所以64位操作系统要用8个字节来保存地址。

问： **如果我创建了一个指针变量，它位于存储器中吗？**

答： 是的，指针变量只不过是一个保存数字的变量罢了。

问： **我可以找到指针变量的地址吗？**

答： 可以用&运算符找到它的地址。

问： **我可以把指针转化为一般的数字吗？**

答： 在大多数操作系统中，可以这样做。C编译器通常会把long数据类型设为和存储器地址一样长。如果想要把指针p保存在long变量a中，可以输入a=(long)p，过几章我们会学习这种方法。

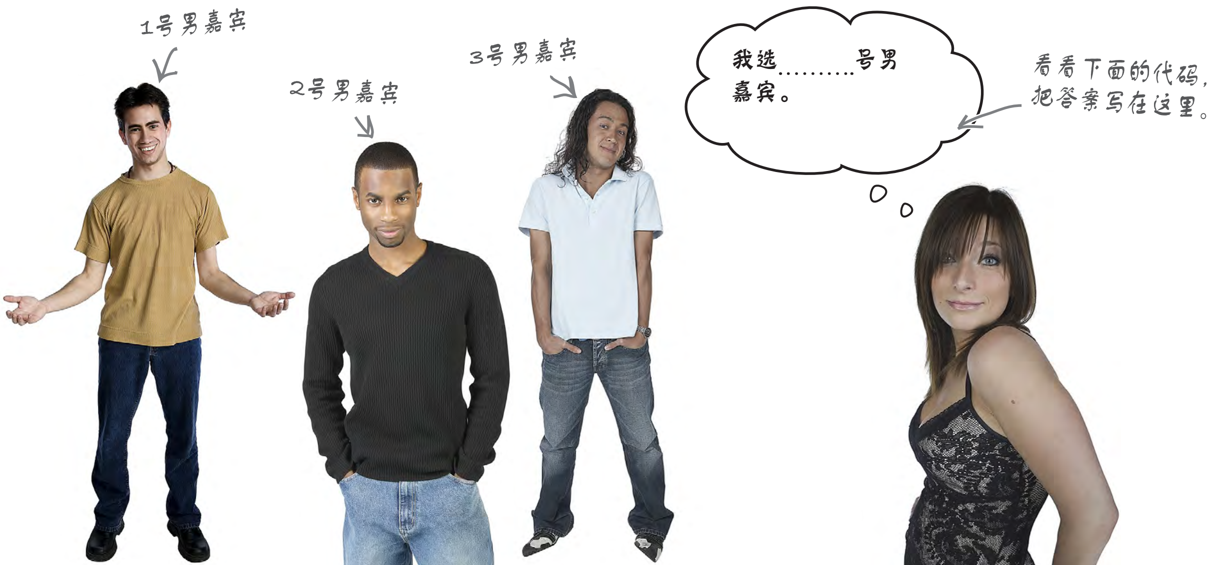
问： **是在大多数操作系统中吗？所以并不是全部？**

答： 并不是全部。

非诚勿扰

三位钻石王老五准备参加今天的“非诚勿扰”。

今晚的幸运女嘉宾将从三位选手中选出她的白马王子，她会选谁呢？

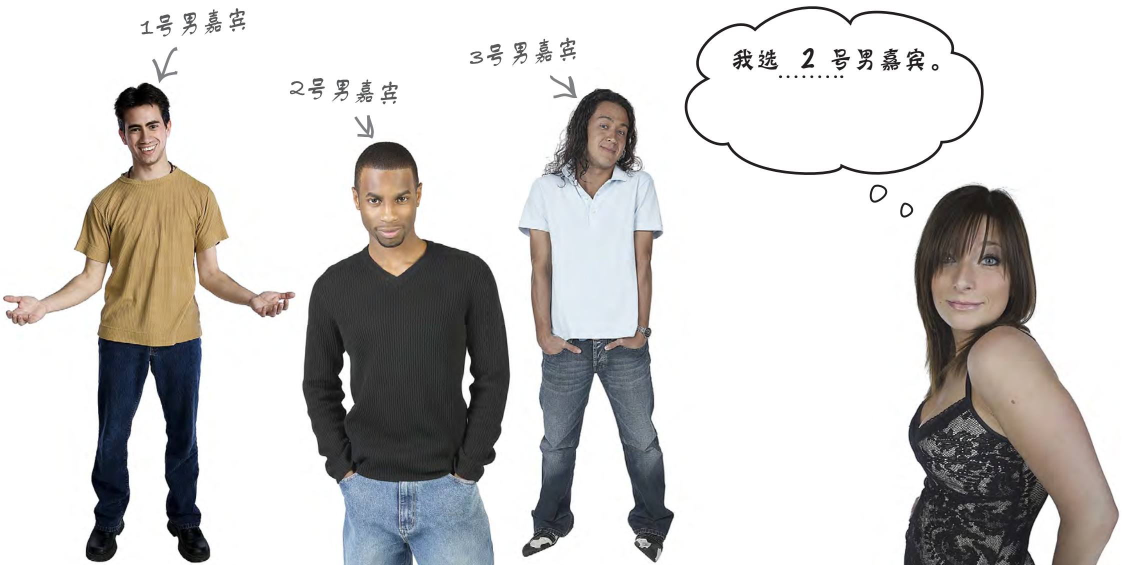



```
#include <stdio.h>

int main()
{
  int contestants[] = {1, 2, 3};
  int *choice = contestants;
  contestants[0] = 2;
  contestants[1] = contestants[2];
  contestants[2] = *choice;
  printf("我选 %i 号男嘉宾", contestants[2]);
  return 0;
}
```

非诚勿扰解答

三位钻石王老五准备参加今天的“非诚勿扰”。

今晚的幸运女嘉宾将从三位选手中选出她的白马王子，她会选谁呢？




```
#include <stdio.h>

int main()
{
  int contestants[] = {1, 2, 3};
  int *choice = contestants;
  contestants[0] = 2;
  contestants[1] = contestants[2];
  contestants[2] = *choice;
  printf("我选 %i 号男嘉宾", contestants[2]);
  return 0;
}
```

choice现在是contestants数组的地址。

contestants[2]
== *choice
== contestants[0]
== 2

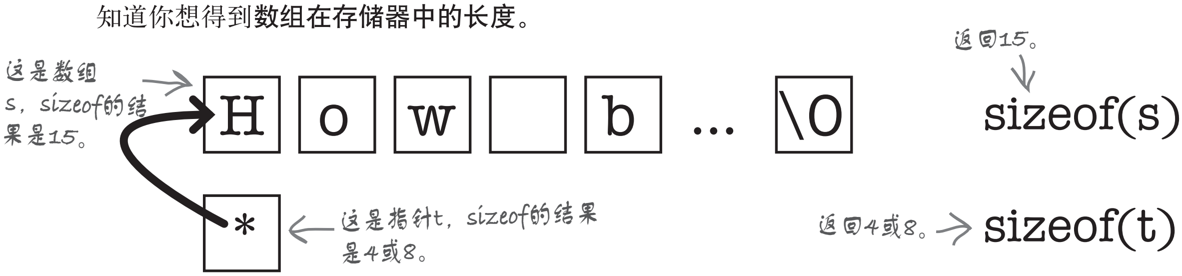
数组变量与指针又不完全相同

虽然可以将数组变量用作指针，但两者还是有一些区别，为了区分它们，考虑下面这段代码：

```
char s[] = "How big is it?";
char *t = s;
```

1. **sizeof（数组）是……数组的大小**

   sizeof(指针)返回了4或8，因为4和8分别是32位和64位操作系统上指针的大小。但如果对数组变量使用sizeof，C语言就开窍了，它知道你想得到**数组在存储器中的长度**。



2. **数组的地址……是数组的地址。**

   指针变量是一个用来保存存储器地址的变量，那数组变量呢？如果对数组变量使用&运算符，结果是数组变量本身。

   &s == s &t != t

   当程序员写下&s时，表示“数组s的地址是？”，数组s的地址就是……s；但如果他写的是&t，则表示“变量t的地址是？”。

3. **数组变量不能指向其他地方。**

   当创建指针变量时，计算机会为它分配4或8字节的存储空间。但如果创建的是数组呢？计算机会为数组分配存储空间，但不会为数组变量分配任何空间，编译器仅在出现它的地方把它替换成数组的起始地址。

   但是由于计算机没有为数组变量分配空间，也就不能把它指向其他地方。

   会报编译错误。→ `s = t;`

指针退化

数组变量和指针变量有一点小小的区别，所以把数组赋给指针时千万要小心。假如把数组赋给指针变量，指针变量只会包含数组的**地址**信息，而对数组的长度一无所知，相当于指针丢失了一些信息。我们把这种信息的丢失称为**退化**。

只要把数组传递给函数，数组免不了退化为指针，但需要记清楚代码中有哪些地方发生过数组退化，因为它们会引发一些不易察觉的错误。

致命处方案件

这栋宅邸拥有他梦寐以求的一切：优美的风景，华丽的吊灯，独立盥洗室。豪宅的主人王百万因心脏病突发死在了花园中，享年94岁。自然死因？医生认为是服用心脏病药物过量。凉亭传来阵阵恶臭，但除了尸体的腐臭好像还有其他的味道。警察走出了大厅，走向王百万的27岁的遗孀，汤茱蒂。

“我想不通，他平时吃药时都很小心，这次怎么会……这张是服药的剂量单。”她将自动服药器的代码拿给他看。

```
int doses[] = {1, 3, 2, 1000};
```

“警察说我改编了服药程序，但我对技术一窍不通。他们说是我写了这段代码，但我认为它不能编译，你说呢？”

她把修过指甲的手伸进钱包，递给了他一份程序。这段代码是警察在百万富翁的床旁边发现的，看起来的确无法编译……

```
printf("服用 %i 毫克的药", 3[doses]);
```

表达式3[doses]是什么意思？3又不是数组。汤茱蒂擤了擤鼻子，说道：“真的不是我写的，况且3毫克的剂量也不算太坏，你说呢？”

3毫克的剂量杀不死那个老男人，但是真相其实近在眼前……

为什么数组从0开始

数组变量可以用作指针，这个指针指向数组的第一个元素，也就是说除了方括号表示法，还可以用*运算符读取数组的第一个元素，像这样：

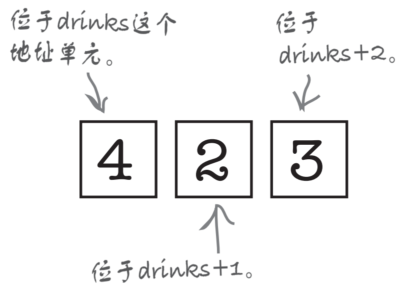
```
int drinks[] = {4, 2, 3};
printf("第一单: %i 杯\n", drinks[0]);
printf("第一单: %i 杯\n", *drinks);
```

这两行代码是等价的。

drinks[0] == *drinks

地址只是一个数字，所以可以进行指针算术运算，比如为了找到存储器中的下一个地址，可以**增加**指针的值。既可以用方括号加上索引值2来读取元素，也可以对第一个元素的地址加2：

```
printf("第三单: %i 杯\n", drinks[2]);
printf("第三单: %i 杯\n", *(drinks + 2));
```



总之，表达式drinks[i]和*(drinks + i)是等价的，这解释了为什么数组要从索引0开始，所谓索引，其实就是为了找到元素的地址单元，指针需要加上的那个数字。

磨笔上阵

使用指针算术运算的魔力修复一颗破碎的心，下面这个函数将跳过文本消息的前六个字符。

```
void skip(char *msg)
{
  puts(..........................);
}

char *msg_from_amy = "Don't call me";
skip(msg_from_amy);
```

为了从第7个字符开始打印这条消息，你需要在这里用什么表达式？

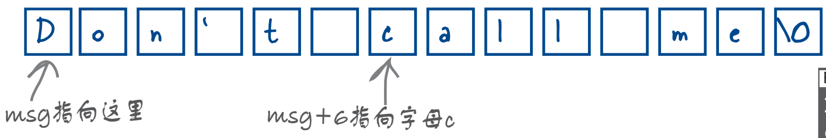
函数需要从字符c开始打印这条消息。

磨笔上阵 解答

你用指针算术运算的魔力修复了一颗破碎的心，下面这个函数跳过了文本消息的前六个字符。

```
void skip(char *msg)
{
  puts( msg +6 );
}

char *msg_from_amy = "Don't call me";
skip(msg_from_amy);
```

msg指针加6，将从第7个字符开始打印这条消息。



代码显示如下。

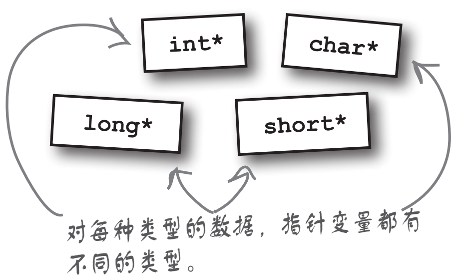
```
File Edit Window Help
> ./skip
call me
>
```

为什么指针有类型

既然指针只是地址，为什么指针变量有类型？为什么不能用一种通用类型的变量保存所有的指针？

因为指针算术运算会暗渡陈仓。如果对char指针加1，指针会指向存储器中下一个地址，那是因为char就占1字节。

如果是int指针呢？int通常占4字节，如果对int指针加1，编译后的代码就会对存储器地址加4。




```
int nums[] = {1, 2, 3};
printf("nums 的地址是 %p\n", nums);
printf("nums + 1 的地址是 %p\n", nums + 1);
```

如果运行上面这段代码，就会发现两个地址相隔不止一个字节。指针之所以有类型，是因为编译器在指针算术运算时需要知道加几。

(nums+1)与nums相隔了4字节。

```
File Edit Window Help
> ./print_nums
nums 的地址是 0x7fff66ccedac
nums + 1 的地址是 0x7fff66ccedb0
```

别忘了，这些地址以16进制的格式打印。

致命处方案件

上次我们把大英雄留在案发现场调查汤茱蒂，她的丈夫因一段可疑的代码而服用药物过量致死。汤茱蒂到底是写代码的杀人凶手，还是无辜的替罪羊？如果你想知道真相，接着往下读……

他把代码塞进了口袋，说道：“王太太，谢谢你的配合，以后我再也不会打扰你了。”他握了握她的手。“谢谢。”她一边拭去眼角的泪水，一边说道，“你真是个好人。”

“先别急，王太太。”还没等汤茱蒂反应过来，他就已经给她戴上了手铐，“看到你那双精心修饰过的手，我就知道你隐瞒了很多真相。”只有一个以键盘为生的人，指尖才会像她那样布满老茧。

“汤茱蒂，你假装对C语言一窍不通，但事实上，你是一个高手，让我们再看一遍代码。”

```
int doses[] = {1, 3, 2, 1000};
printf("服用 %i 毫克的药", 3[doses]);
```

“看到3[doses]这个表达式的时候，我就知道有哪里不对劲，你知道数组变量doses用作指针，那剂致命的1000毫克可以写成这样……”他在一张“舒洁”纸巾上写下一些代码。

```
doses[3] == *(doses + 3) == *(3 + doses) == 3[doses]
```

“王太太，你的代码彻底出卖了你，它给王百万服用了1000毫克的药。现在我们要带你去一个地方，在那里你再也无法玩弄C语言的语法……”

要点

- 数组变量可以用作指针……
- ……但数组变量和指针又不完全相同。
- 对数组变量和指针变量使用sizeof，效果不同。
- 数组变量不能指向其他地方。
- 把数组变量传给指针，会发生退化。
- 索引的本质是指针算术运算，所以数组从0开始。
- 指针变量具有类型，这样就能调整指针算术运算。

这里没有蠢问题

问： **我真的需要理解指针算术运算吗？**

答： 有的程序员不使用指针算术运算，因为它很容易出错，但你可以用它有效地处理数组数据。

问： **指针能做减法吗？**

答： 能，但小心别让指针越过数组的起点。

问： **C语言什么时候对指针算术运算进行调整？**

答： 在编译器生成可执行文件时，编译器会根据变量的类型，用变量的大小乘以指针的增量或减量。

问： **然后呢？**

答： 假如编译器看到你对一个指向int数组的指针加2，就会用2乘以4（int的长度），然后对地址加8。

问： **C语言在调整指针算术运算时用了sizeof运算符吗？**

答： 本质上如此，sizeof运算符的结果也是在编译时决定的，对各种数据类型，sizeof和指针算术运算都将使用相同的长度。

问： **指针可以相乘吗？**

答： 不可以。

用指针输入数据

你已经知道了怎样让用户从键盘输入字符串，可以用scanf()函数：

你将把人名保存在这个数组中。

```
char name[40];
printf("Enter your name: ");
scanf("%39s", name);
```

scanf总共会读取39个字符，以及字符串终结符\0。

scanf()是怎么工作的呢？它接收一个char指针，而在这个例子中，传给了它一个数组变量。这时你一定在想**为什么**scanf()要接收指针，这是因为scanf()函数打算更新数组的内容，一个想要更新变量的函数可不需要变量本身的值，它要的是变量的**地址**。

用scanf()输入数字

怎么把数据输进数值字段呢？传递一个指向数值变量的指针就行了。

```
int age;
printf("Enter your age: ");
scanf("%i", &age);
```

%i表示用户会输入一个int值。

用&运算符得到int的地址。

把数值变量的地址传进了函数，scanf()便可以更新变量的内容。为了帮你排忧解难，scanf()允许传递格式字符串，就像你对printf()函数做的那样，甚至可以用scanf()一次输入多条信息：

%i ← 输入一个整数。

%29s ← 输入29个字符（+‘\0’）。

%f ← 输入一个浮点数。

```
char first_name[20];
char last_name[20];
printf("Enter first and last name: ");
scanf("%19s %19s", first_name, last_name);
printf("First: %s Last:%s\n", first_name, last_name);
```

读取名，接着是一个空格，然后是姓。

名和姓分别保存在两个数组中。

```
File Edit Window Help Meerkats
> ./name_test
Enter first and last name: Sanders Kleinfeld
First: Sanders Last: Kleinfeld
>
```

使用scanf()时要小心

scanf()函数有一个小毛病。到目前为止，你写过的所有代码都小心翼翼地限制了scanf()能读入的字符数：

```
scanf("%39s", name);

scanf("%2s", card_name);
```

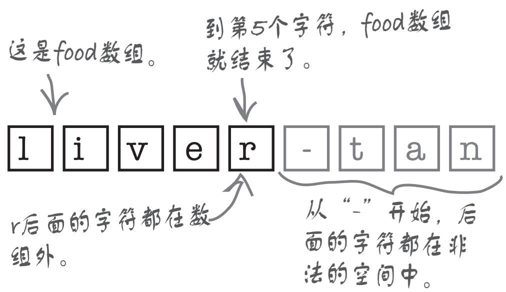
为什么要这样做？毕竟scanf()用了和printf()一样的格式串，但当我们用printf()打印字符串时，只用了%s。如果在scanf()中只用%s，一旦用户输入得太起劲，就会出问题：

```
char food[5];
printf("Enter favorite food: ");
scanf("%s", food);
printf("Favorite food: %s\n", food);
```

File Edit Window Help TakeAByte

```
> ./food
Enter favorite food: liver-tangerine-raccoon-toffee
Favorite food: liver-tangerine-raccoon-toffee
Segmentation fault: 11
>
```

程序崩溃了，因为scanf()在写数据时越过了food数组的尾部。



scanf()会导致缓冲区溢出

如果忘了限制scanf()读取字符串的长度，用户就可以输入远远超出程序空间的数据，多余的数据会写到计算机还没有分配好的存储器中。如果运气好，数据不但能保存，而且不会有任何问题。

但缓冲区溢出很有可能会导致程序出错，这种情况通常被称为段错误或abort trap，不管出现什么错误消息，程序都会**崩溃**。

除了scanf()还可以用fgets()

还可以用另一个函数来输入文本数据：**fgets()**。和scanf()函数一样，fgets()接收char指针，不同的是，你必须给出最大长度：

这个程序与之前的一样。

```
char food[5];
printf("Enter favorite food: ");
fgets(food, sizeof(food), stdin);
```

首先，它接收指向缓冲区的指针。

其次，它接收字符串（包括“\0”）的最大长度。

stdin表示数据将来自键盘。

稍后会看到关于stdin的更多内容。

也就是说当调用fgets()时不可能一不小心忘记设置长度，因为它就出现在了函数的签名中，所以不得不加这个参数。另外，注意fgets()缓冲区大小把\0字符也算了进去，所以不必像scanf()那样把长度减1。

关于fgets()，还需要知道什么？

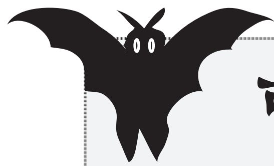
fgets()配合sizeof一起使用

上面这段代码用sizeof运算符设置了最大长度，小心，别忘了sizeof返回变量占用空间的大小。在上面这段代码中，food是数组变量，所以sizeof返回了数组的大小；如果food是指针变量，sizeof仅仅会返回指针的大小。

如果你要向fgets()函数传递数组变量，就用sizeof；如果只是传指针，就应该输入你想要的长度。

如果food是一个指针，就不能用sizeof，而应该显式给出长度。

```
printf("Enter favorite food: ");
fgets(food, 5, stdin);
```



古墓迷案

fgets()函数其实是从一个更古老的函数演变而来的，它叫gets()。

尽管我们说fgets()比scanf()更加安全，但它的祖先gets()才是最危险的家伙。为什么？因为gets()函数没有任何限制：

别！我是认真的，千万别用它。

```
char dangerous[10];
gets(dangerous);
```

虽然gets()函数已经行走江湖很多年了，但真的不应该用它。

拳王争霸赛

醒醒！醒醒！我们期待已久的拳王争霸赛现在开始。身披红色战袍的是身轻如燕、灵活机动但是有一点点危险的数据输入坏小子：scanf()。蓝方朴实无华、安全可靠，你一定想把他介绍给你妈妈：他就是fgets()！

	scanf():	**fgets():**
第1回合：限制 限制用户输入的字符数吗？	只要记得在格式串中加入长度，scanf()就能限制用户输入数据的长度。	fgets()强行限制用户输入字符串的长度，可谓无懈可击。

结果：fgets()凭点数胜出。

第2回合：多字段 能输入多个字段吗？	能！scanf()不但允许输入多个字段，而且允许输入结构化数据，可以指定两个字段之间以什么字符分割。	哎呦！fgets()遭到了一次迎头痛击。fgets()只允许向缓冲区中输入一个字符串，而且只能是字符串，不能是其他数据类型，只能有一个缓冲区。

结果：scanf()完胜。

第3回合：字符串中的空格 用户能输入带空格的字符串吗？	呜！scanf()被这一拳头打得够呛。当scanf()用%s读取字符串时，遇到空格就会停止。如果想要输入多个单词，需要多次调用scanf()，或使用一些复杂的正则表达式技巧。	小菜一碟，无论何时，fgets()都能读取整个字符串。

结果：绝地大反击！fgets()拿下了这一回合。

这两个急脾气的函数之间展开了一场干净漂亮的决斗。显然，如果需要输入由多个字段构成的**结构化数据**，可以使用scanf()；而如果想要输入一个**非结构化的字符串**，fgets()将是你的不二之选。

三猜一

在Head First酒吧的地下室中，有人在玩“三猜一”。某人不停地交换三张牌的位置，你必须屏息凝视，指出Q去了哪儿。当然，在Head First酒吧中，他们并没有用真牌，而是用了代码。下面是他们所使用的程序：

```
#include <stdio.h>

int main()
{
  char *cards = "JQK";
  char a_card = cards[2];
  cards[2] = cards[1];
  cards[1] = cards[0];
  cards[0] = cards[2];
  cards[2] = cards[1];
  cards[1] = a_card;
  puts(cards);
  return 0;
}
```

找到Q。

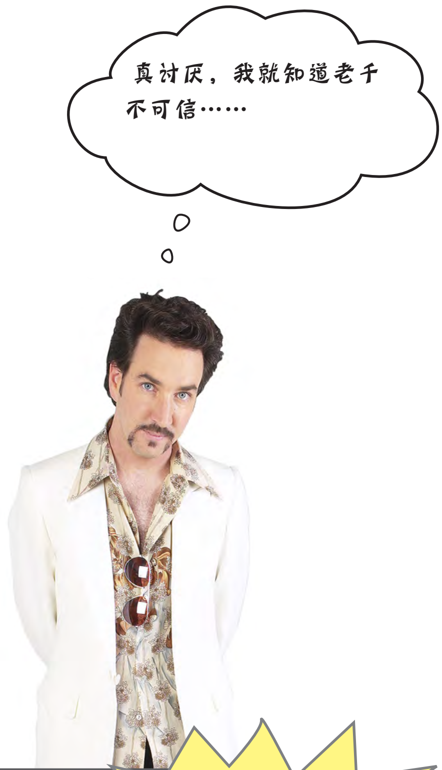
代码旨在交换字符串“JQK”中的三个字母。别忘了，在C语言中，字符串只是一个字符数组。程序不停交换字符的位置，最后显示字符串。

玩家把钱押到他们认为是Q的那个数组元素上，然后编译并运行代码。

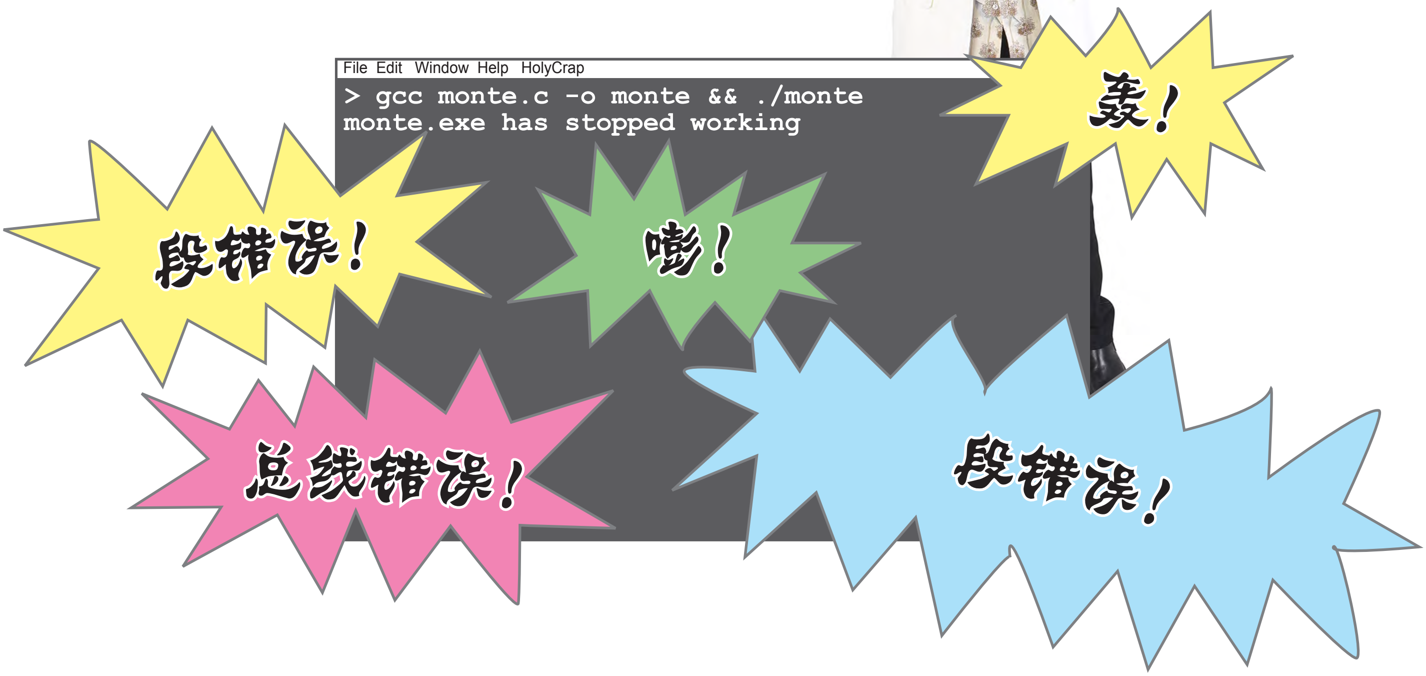
哎呀……存储器故障……

看来老千的代码出了一点问题，当代码在酒吧的笔记本电脑上编译、运行时，发生了：

```
File  Edit  Window  Help  PlaceBet
> gcc monte.c -o monte && ./monte
bus error
```



而且，当这帮人把同一段代码放到不同计算机和不同操作系统上编译并运行时，得到了一大堆不同的错误：



代码错在哪里？

心灵的呼唤

是时候使用你的直觉了，别想太多，但猜无妨。请**通读**以下候选答案，选出你认为正确的那一个。

你认为问题出在哪里？

字符串无法更新。	
我们把字符交换到了字符串的外面。	
字符串不在存储器中。	
其他原因。	

心灵的呼唤 解答

是时候使用你的**直觉**了。你通读了以下候选答案，选出了你认为正确的那一个。

你认为问题出在哪里？

字符串无法更新。	✓
我们把字符交换到了字符串的外面。	
字符串不在存储器中。	
其他原因。	

字符串字面值不能更新

指向字符串字面值的指针变量不能用来修改字符串的内容：

```
char *cards = "JQK";
```

← 不能用这个变量修改这个字符串。

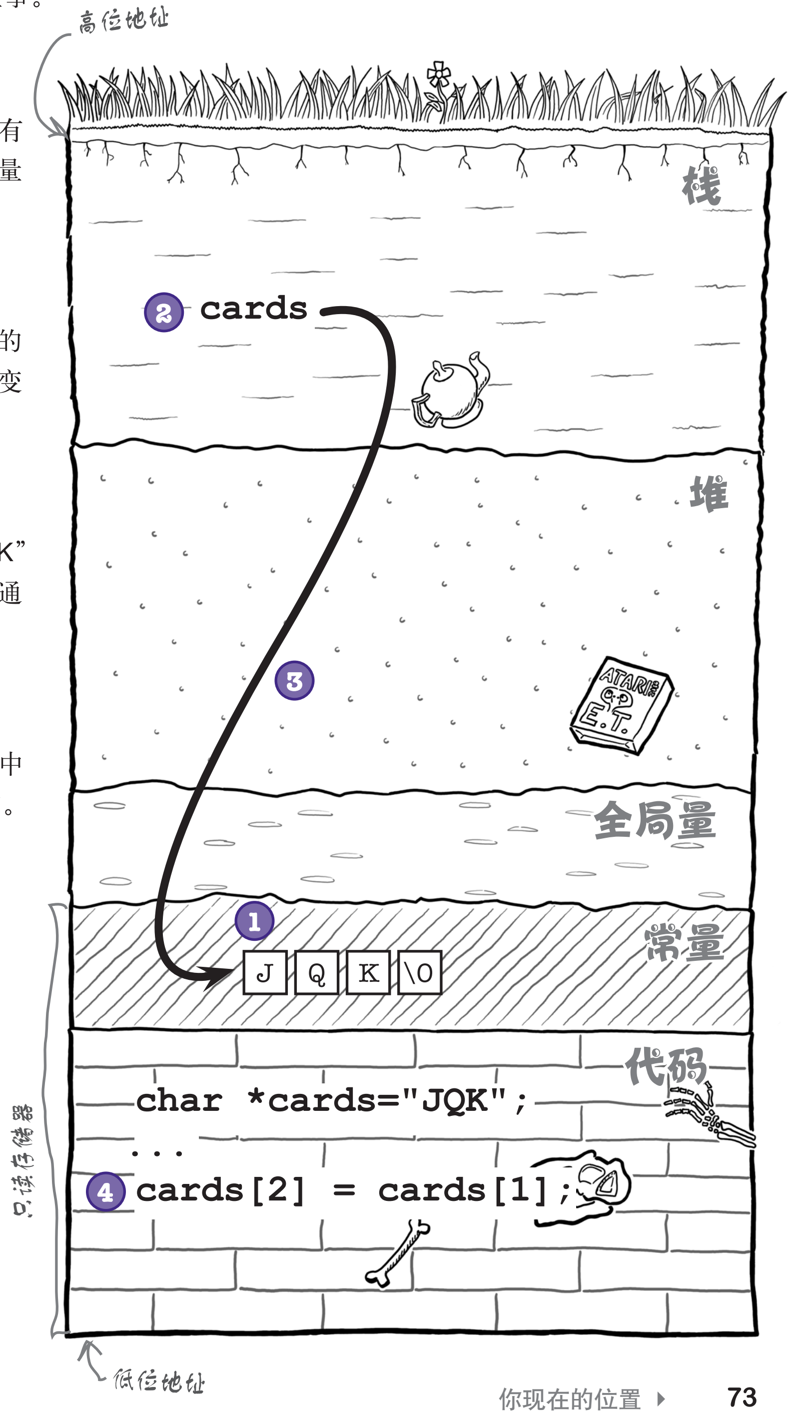
但如果你用字符串字面值创建一个数组，就**可以**修改了：

```
char cards[] = "JQK";
```

这是由C语言使用存储器的方式决定的……

存储器中的char *cards=“JQK”；

为了弄明白这行代码导致存储器出错的原因，我们需要切开计算机的存储器，看看计算机究竟将会做哪些事。

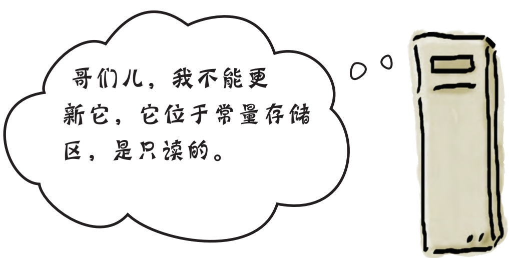


1. **计算机加载字符串字面值。**
当计算机把程序载入存储器时，会把所有常数值（如字符串常量“JQK”）放到常量存储区，这部分存储器是**只读**的。

2. **程序在栈上创建cards变量。**
栈是存储器中计算机用来保存局部变量的部分，局部变量也就是位于函数内部的变量，`cards`变量就在这个地方。

3. **cards变量设为“JQK”的地址。**
`cards`变量将会保存**字符串字面值**“JQK”的地址。为了防止修改，字符串字面值通常保存在只读存储器中。

4. **计算机试图修改字符串。**
程序试图修改`cards`变量指向的字符串中的内容时就会失败，因为字符串是只读的。



所以问题出在像“JQK”这样的字符串字面值保存在只读存储器中。它们是常量。

既然知道了症结所在，如何对症下药呢？

如果想修改字符串，就复制它

如果想要修改字符串的内容，就需要对它的**副本**进行操作。如果在存储器的非只读区域创建了字符串的副本，就可以修改它的字母了。

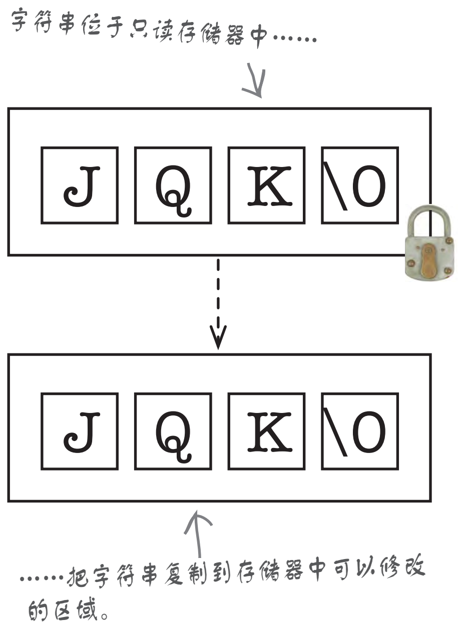
怎么创建副本？用字符串创建一个新数组就行了。

```
char cards[] = "JQK";
```

cards不只是指针，cards现在是数组。

为什么这个方法行得通呢？一切字符串皆为数组，但是在旧代码中，cards只是一个指针，而在新代码中，它是一个数组。假如你声明了一个名为cards的数组，然后把它设置成字符串字面值，cards数组就成为了一个全新副本。cards不再只是一个指向字符串字面值的指针，而是一个崭新的数组，里面保存了字符串字面值的最新副本。

为了弄明白它实际是怎么工作的，需要看看存储器中发生了什么。



百宝箱

cards[]还是*cards?

如果看到这样的声明，会觉得它是什么意思？

```
char cards[]
```

这取决于在什么地方看到它，如果是普通的变量声明，cards就是一个数组，而且必须立即赋值：

```
int my_function()
{
  char cards[] = "JQK";
  ...
}
```

cards是数组。

因为没有给出数组的大小，必须立即赋值。

但如果cards以函数参数的形式声明，那么cards就是一个指针：

```
void stack_deck(char cards[])
{
  ...
}
```

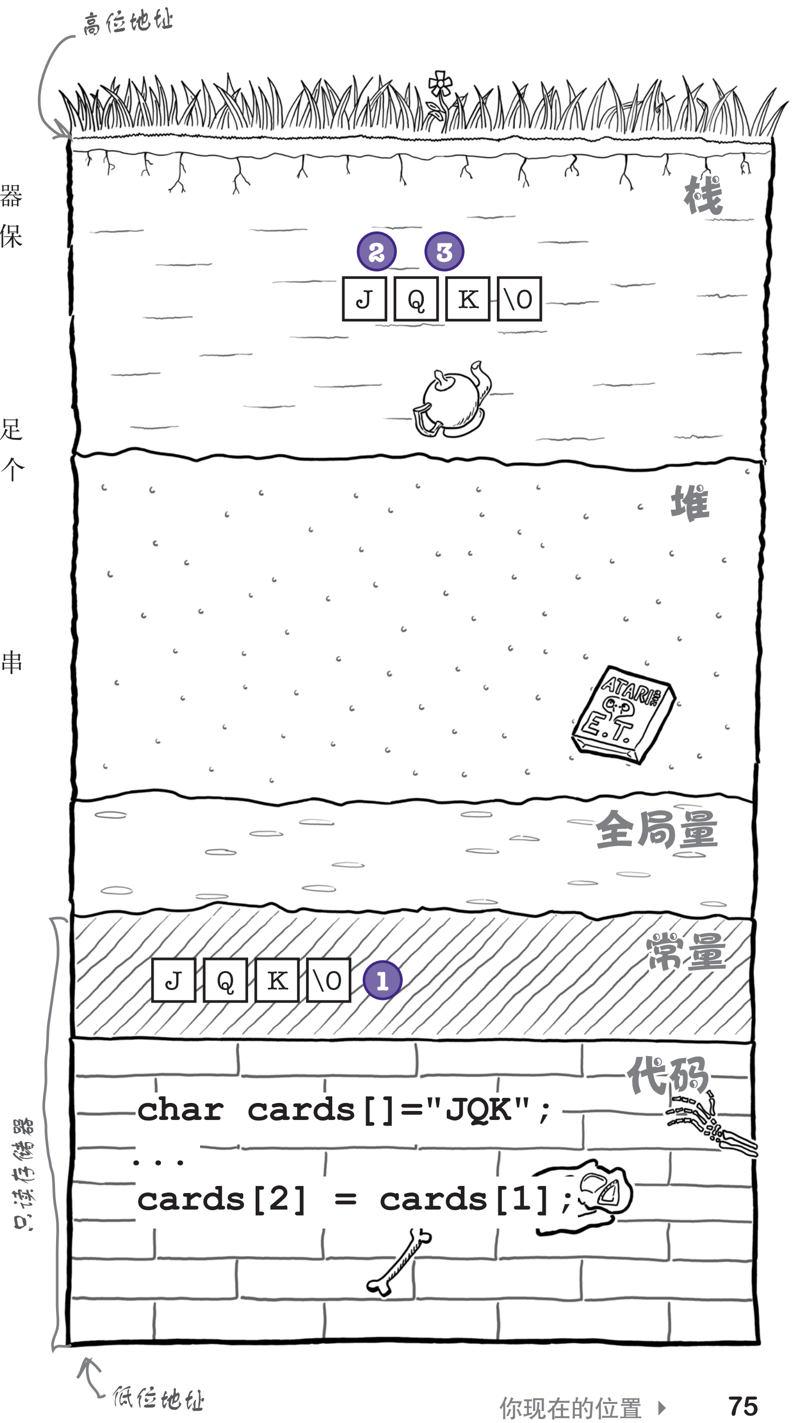
cards是char指针。

```
void stack_deck(char *cards)
{
  ...
}
```

这两个函数是等价的。

存储器中的char cards[]= “JQK”；

我们已经见过了那段有问题的代码在运行时计算机做了哪些事。那么新代码呢？我们来瞧瞧。

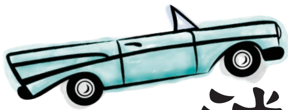


1 **计算机载入字符串字面值。**
和刚刚一样，当计算机把程序载入存储器时，会把常量值（如字符串“JQK”）保存到只读存储器。

2 **程序在栈上新建了一个数组。**
我们声明了数组，所以程序会创建一个足够大的数组来保存字符串“JQK”，在这个例子中4个字符足矣。

3 **程序初始化数组。**
除了为数组分配空间，程序还会把字符串字面值“JQK”的内容复制到栈上。

区别是原来的代码使用了指向只读字符串字面值的指针；而在新的代码中，你用字符串字面值初始化了一个数组，从而得到了这些字母的副本，这样就可以随意修改它们了。



试驾

在代码中构造一个新数组，看看会发生什么。

```
#include <stdio.h>

int main()
{
  char cards[] = "JQK";
  char a_card = cards[2];
  cards[2] = cards[1];
  cards[1] = cards[0];
  cards[0] = cards[2];
  cards[2] = cards[1];
  cards[1] = a_card;
  puts(cards);
  return 0;
}
```

```
File Edit Window Help Where'sTheLady?
> gcc monte.c -o monte && ./monte
QKJ
```

代码正确工作了！ cards变量现在指向存储器中非只读区域中的字符串，所以我们可以自由地修改它的内容。

百宝箱

为了从此避免这个错误，可以不再将char指针设置为字符串字面值，像这样：

```
char *s = "Some string";
```

但是把指针设为字符串字面值又没错，问题出在你试图修改字符串字面值。如果你想把指针设成字符串字面值，必须确保使用了const关键字：

```
const char *s = "some string";
```

这样一来，如果编译器发现有代码试图修改字符串，就会提示编译错误：

```
s[0] = 'S';
monte.c:7: error: assignment of read-only location
```

神奇子弹案件

他正在向音乐管理软件中导入“枪炮玫瑰”乐队的全集，突然听到有人敲门。一个女人走了进来，她身高1米68，金色的头发，挎了一个别致的笔记本包，穿了一双便宜的鞋。他有很强烈的预感，她是个女程序员。“求你帮帮我……你一定要洗刷他的清白，我可以保证，Jimmy是无辜的！”他递给她一张纸巾让她擦掉眼泪，请她就坐。

和所有老掉牙的故事一样，故事的开头是：她遇到了一个男人。Jimmy Blomstein是当地一家星巴克的服务生，他的业余爱好是骑自行车与收集动物标本，他梦想着有一天能制作一头大象标本。可惜他交友不慎。那天早上，蒙面劫匪Masked Raider去喝咖啡时遇到了Jimmy，那时他们还活着：

```
char masked_raider[] = "Alive";
char *jimmy = masked_raider;
   printf("Masked raider is %s, Jimmy is %s\n", masked_raider,
   jimmy);
```

五分钟推理剧

```
File Edit Window Help
Masked raider is Alive, Jimmy is Alive
```

那天下午，蒙面劫匪动身去抢劫酒吧。过去的一百次行动他都未曾失手，但这一次，当他一脚踹开Head First酒吧地下室的门时，却发现联邦调查局的人正在玩“三猜一”欢度周末。枪声响起，一声尖叫，歹徒倒在人行道上，人群一阵骚乱。

```
masked_raider[0] = 'D';
masked_raider[1] = 'E';
masked_raider[2] = 'A';
masked_raider[3] = 'D';
masked_raider[4] = '!';
```

奇怪的是，当这个女人去咖啡店找他男朋友时，却被告知昨晚他调的那杯橙色摩卡冰乐已成了绝唱。

```
printf("Masked raider is %s, Jimmy is %s\n", masked_raider, jimmy);
```

```
File Edit Window Help
Masked raider is DEAD!, Jimmy is DEAD!
```

到底发生了什么？为什么一颗神奇的子弹同时杀死了Jimmy和蒙面劫匪？你认为发生了什么？

神奇子弹案件

为什么一颗神奇的子弹同时杀死了Jimmy和蒙面劫匪？

在大恶魔蒙面劫匪被击毙的一瞬间，Jimmy，这位温文尔雅的咖啡师也被离奇击中：

```
#include <stdio.h>
int main()
{
  char masked_raider[] = "Alive";
  char *jimmy = masked_raider;
  printf("Masked raider is %s, Jimmy is %s\n", masked_raider, jimmy);
  masked_raider[0] = 'D';
  masked_raider[1] = 'E';
  masked_raider[2] = 'A';
  masked_raider[3] = 'D';
  masked_raider[4] = '!';
  printf("Masked raider is %s, Jimmy is %s\n", masked_raider, jimmy);
  return 0;
}
```

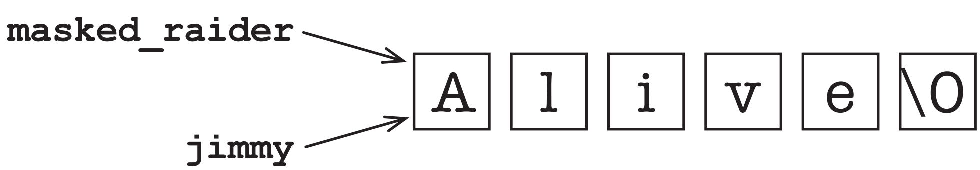
市场部友情示：广告费没谈拢，去掉这里的健脑饮料植入广告。

侦探花了很长时间才听她讲完这个故事，整个过程中，他不断喝着手中的Head First健脑水果饮料。他坐回椅子，朝那双蓝色的眼睛望去，她就像一辆疾驰的卡车前的兔子，灯光已经笼盖了她的身体，命在顷刻，而他就是那个紧握方向盘的人。

“恐怕我要告诉你一个坏消息，Jimmy和蒙面劫匪……是同一个人。”

“怎么可能！”

她猛吸一口气，用手捂住了嘴。“抱歉，女士，但我必须告诉你我是怎么看出来的。来看看存储器的使用情况。”他画了一个图。



“jimmy和masked_raider是同一个存储器地址的两个别名，它们都指向了同一个地方。当masked_raider被子弹击中时，Jimmy也无法幸免。这里还有一张发票，抬头是旧金山大象保护中心，这里还有一张15吨包装材料的订单，铁证如山，我想真相已经大白。”

要点

- 如果在变量声明中看到*，说明变量是指针。
- 字符串字面值保存在只读存储器中。
- 如果想要修改字符串，需要在新的数组中创建副本。
- 可以将char指针声明成为const char *，以防代码用它修改字符串。

这里没有蠢问题

问：为什么编译器不直接告诉我“不能修改字符串”？

答：我们只是把cards声明成char *，编译并不知道这个变量指向字符串字面值。

问：为什么字符串字面值要保存在只读存储器中？

答：因为它们专门用来表示常量。如果你写了一个打印“Hello World”的函数，一定不想让程序的其他部分修改“Hello World”这个字符串字面值。

问：只读变量在所有操作系统中都不能够修改吗？

答：大部分操作系统都有这个规定，一些Cygwin的gcc版本允许修改字符串字面值，不会报错，但这样做常常是错的。

问：const到底是什么意思？它能让字符串变成只读吗？

答：加不加const，字符串字面值都是只读的，const修饰符表示，一旦你试图用const修饰过的变量去修改数组，编译器就会报错。

问：在存储器中，不同的存储器段总是以相同的顺序出现吗？

答：在同一种操作系统中它们出现的顺序相同，不同操作系统之间略有差异，例如Windows的代码段就不在地址的低位。

问：我还是不理解，为什么数组变量不保存在存储器中？既然它存在，就应该在某个地方，不是吗？

答：程序在编译期间，会把所有对数组变量的引用替换成数组的地址。也就是说在最后的可执行文件中，数组变量并不存在。既然数组变量从来不需要指向其他地方，有和没有其实都一样。

问：每当我把一个新数组设为字符串字面值，程序实际上会复制字符串字面值的内容吗？

答：这取决于编译器，最后的机器代码既有可能把整个字符串字面值的内容复制到数组，也有可能程序会根据声明设置每个字符的值。

问：你老是在说“声明”，它是什么意思？

答：声明是一段代码，它声称某样东西（变量或函数）存在；而定义说明它是什么东西。如果在声明了变量的同时将其设为某个值（例如int x = 4;），这段代码既是声明又是定义。

问：为什么scanf()要被称为scanf()？

答：scanf()其实表示“scan formatted”，它用来扫描带格式的输入。

把存储器保存在大脑里

栈

这是存储器用来保存**局部变量**的部分。每当调用函数，函数的所有局部变量都在栈上创建。它之所以叫栈是因为它看起来就像堆积而成的栈板：当进入函数时，变量会放到栈顶；离开函数时，把变量从栈顶拿走。奇怪的是，栈做起事来颠三倒四，它从存储器的顶部开始，向下增长。

堆

我们还没有用过这部分的存储器，堆用于**动态存储**：程序在运行时创建一些数据，然后使用很长一段时间，稍后会看到堆的用法。

全局量

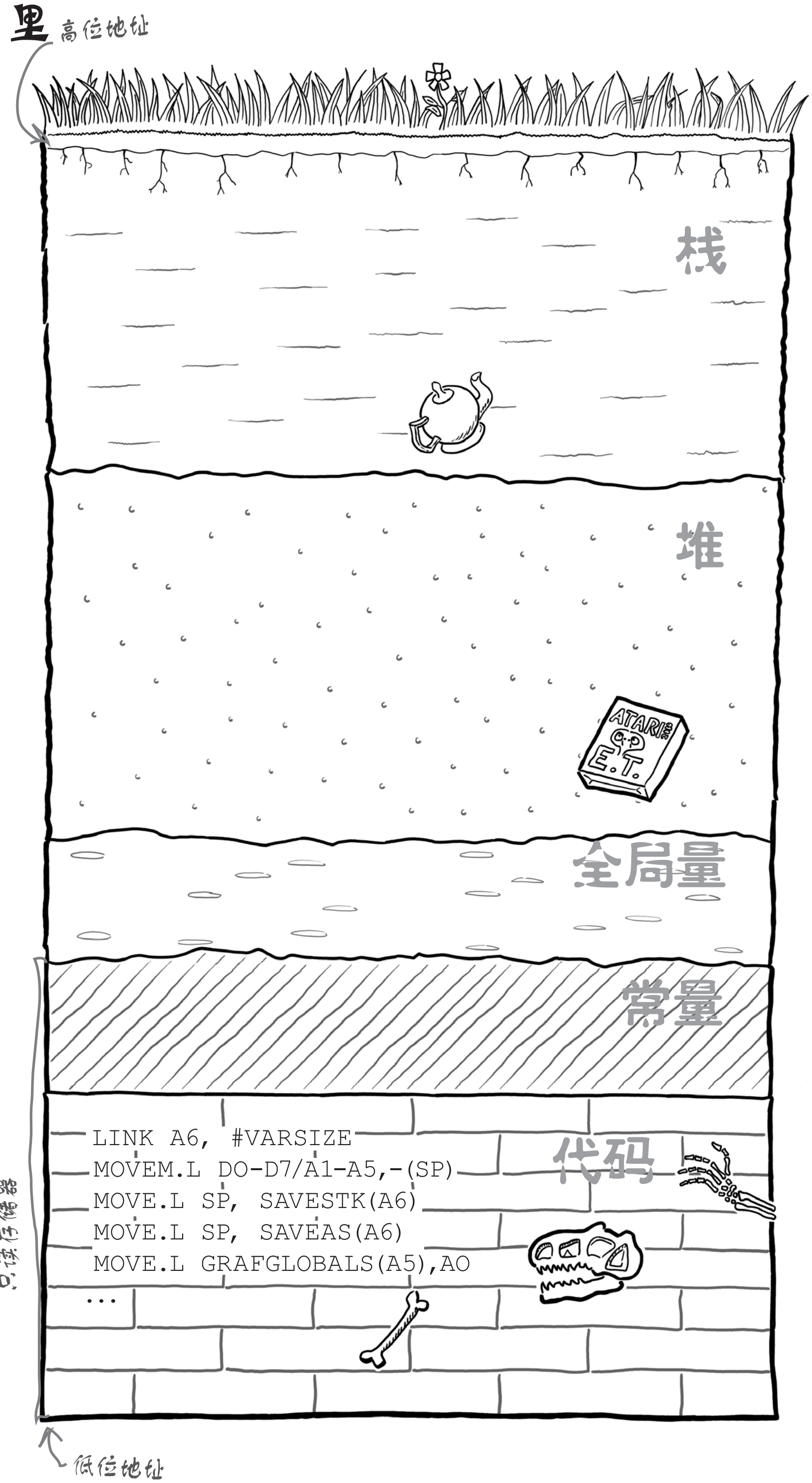
全局量位于所有函数之外，并对所有函数可见。程序一开始运行时就会创建全局量，你可以修改它们，不像……

常量

常量也在程序一开始运行时创建，但它们保存在只读存储器中。常量是一些在程序中要用到的不变量，你不会想修改它们的值，例如字符串字面值。

代码

最后是代码段，很多操作系统都把代码放在存储器地址的低位。代码段也是只读的，它是存储器中用来加载机器代码的部分。



C语言工具箱

你已经学完了第2章，现在你的工具箱又加入了指针和存储器。关于本书提示工具条的完整列表，请见附录ii。

scanf("%i", &x)可以让用户直接输入数字x。

不同计算机的int的大小不同。

&x返回x的地址。

&x称为指向x的指针。

char指针变量x可以这么声明：char *x。

局部变量保存在栈上。

字符串字面值保存在只读存储器中。

数组变量可以用作指针。

用字符串初始化数组，数组会复制字符串中的内容。

用*a读取地址a中的内容。

可以简单地用fgets(buf, size, stdin)输入文本。

2.5 字符串

字符串原理

字符串不只是用来读取的。

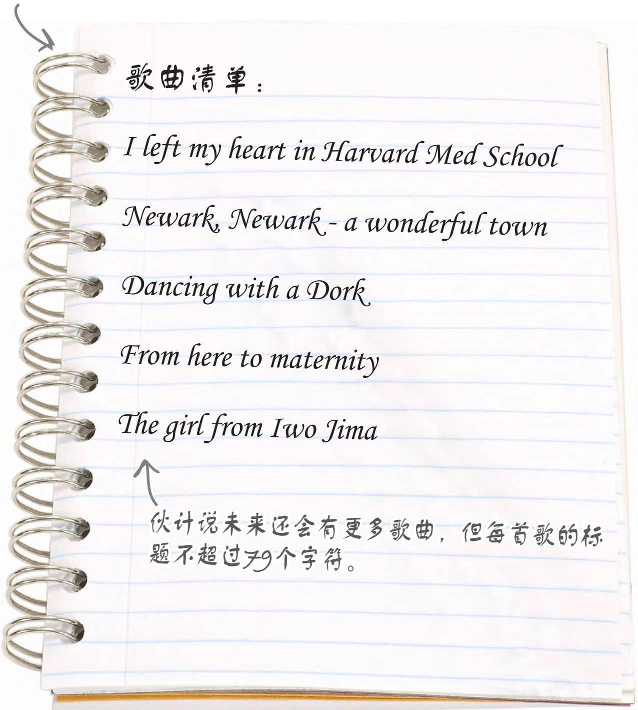
在C语言中字符串其实就是`char`数组，这你已经知道了，问题是字符串能用来干嘛？该*string.h*出场了。*string.h*是C标准库的一员，它负责处理字符串。如果想要连接、比较或复制字符串，*string.h*中的函数就可以派上用场了。在本章中，你将学会如何创建字符串数组，并近距离观察如何使用**`strstr()`**函数搜索字符串。

不顾一切找~~Susan~~ Frank

老式自动点唱机上的歌太多了，人们找不到他们想要听的。为了帮助顾客，Head First酒吧的伙计希望你再写一个程序。

歌曲清单如下：

新专辑《鲜为人知的Sinatra》[①]中的歌曲。



这只是前面几首歌，这张歌单很有可能变长。你需要写一个C程序，要求用户输入想找的歌名，搜索所有的歌曲，显示匹配的曲目。

脑力风暴

程序中有很多字符串，怎样用C语言记录这些信息呢？

① Frank Sinatra和猫王、Wayne Newton是同一时期的著名美国艺人。这里提到的几首歌曲均改头换面自他演唱过的歌曲：*I Left My Heart In San Francisco*、*New York, New York – A Wonderful Town*、*Dancing In A Dark*、*From Here To Eternity*、*The Girl From Ipanema*。——译者注

创建数组的数组

你需要记录几首歌的名字。可以一次用数组记录几件事情。但别忘了，字符串本身就是一个数组，也就是说需要创建数组的数组，像这样：

第一对方括号用来访问由所有字符串组成的数组。

第二对方括号用来访问每个单独的字符串。

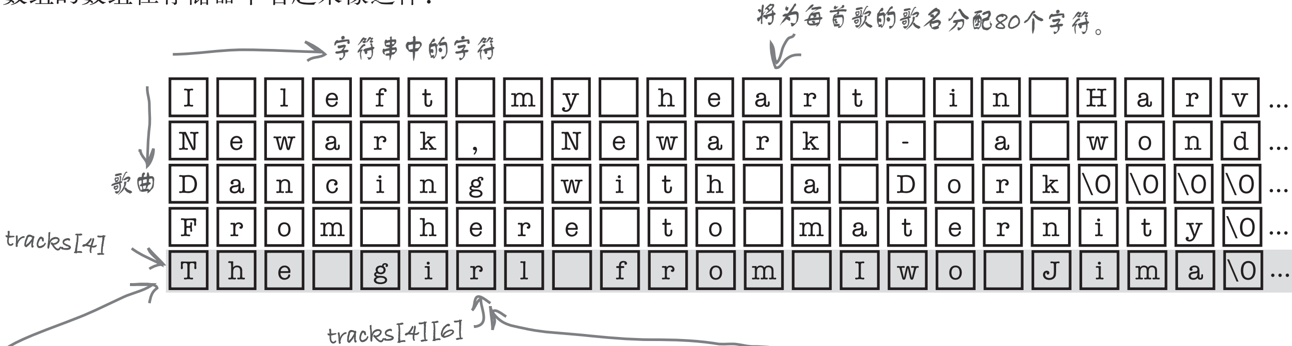
编译器可以识别你有5个字符串，因此括号里可以不写数字。

歌名不得超过79个字符，所以将这个值设为80。

```
char tracks[][80] = {
  "I left my heart in Harvard Med School",
  "Newark, Newark - a wonderful town",
  "Dancing with a Dork",
  "From here to maternity",
  "The girl from Iwo Jima",
};
```

每个字符串都是一个数组，所以这是一个数组的数组。

数组的数组在存储器中看起来像这样：



也就是说，为了找到某一首歌的名字，可以这样写：

它的值是这个。　这是第5个字符串。←别忘了，数组从0开始。

tracks[4] ⟶ "The girl from Iwo Jima"

如果想读取字符串中的某个字符：

tracks[4][6] ⟶ 'r' ← 这是第5个字符串中第7个字符。

现在你知道如何用C语言记录这些数据，问题是该如何使用这些数据？

找到包含搜索文本的字符串

酒吧的那伙人给了你一个程序说明书，真是雪中送炭。

> 让用户输入要找的歌曲。
> 循环遍历所有歌名。
> 如果歌名包含搜索文本，
> 就显示出来。

好了，你知道怎么记录歌名，自然也会读取歌名，循环遍历所有歌名对你来说也不是什么难事，你甚至还能让用户输入想要搜索的文本。唯一的问题是怎么判断歌名中是否含有某段文本。

使用string.h

当安装C编译器时可以免费得到一批很有用的代码——C**标准库**。标准库中的代码能做很多有用的事情，例如打开文件、做算术以及管理存储器，但不可能一次用到整个标准库，因此标准库分了好几个部分，每个部分都有一个**头**文件，列出了这部分标准库中的所有函数。

到目前为止，你只用过*stdio.h*头文件。*stdio.h*提供了标准输入/输出函数，如`printf`和`scanf`。

标准库也含有处理字符串的代码。很多程序都需要处理字符串，而标准库中用来处理字符串的代码不但久经考验、稳定可靠，速度还很快。

你可以用***string.h***头文件把处理字符串的代码包含到程序中，把*string.h*加在程序的顶端，就像你包含*stdio.h*那样。

```
#include <stdio.h>
#include <string.h>
```

将在点唱机程序中同时使用stdio.h和string.h。

猜猜我是谁？

你能把下列*string.h*函数和它们的作用连起来吗？

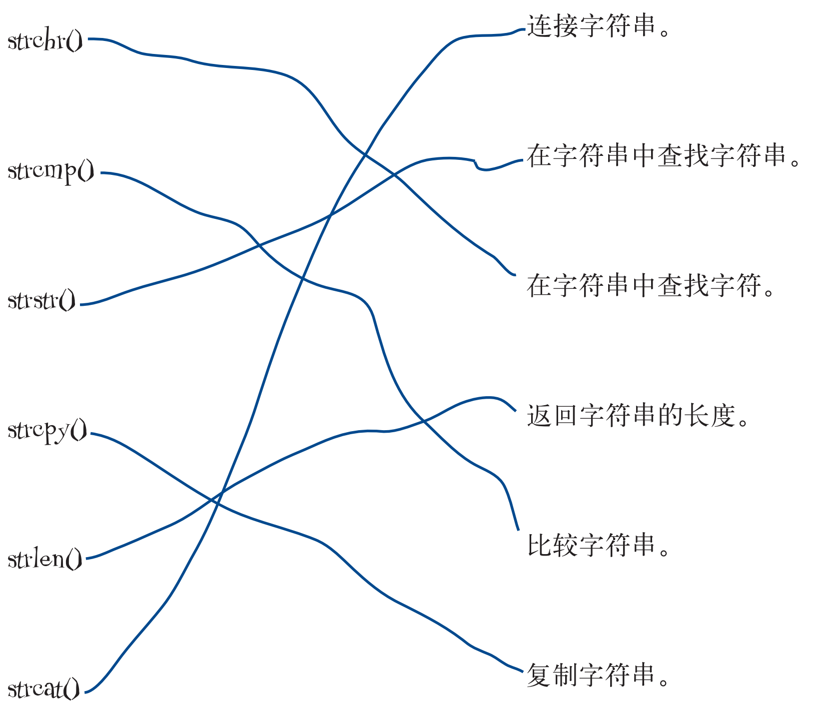
strchr()	连接字符串。
strcmp()	在字符串中查找字符串。
strstr()	在字符串中查找字符。
strcpy()	返回字符串的长度。
strlen()	比较字符串。
strcat()	复制字符串。

磨笔上阵

应该在点唱机程序中使用哪个函数？请把答案写在下面。

猜猜我是谁？解答

你将把下列*string.h*函数和它们的作用连起来。



磨笔上阵解答

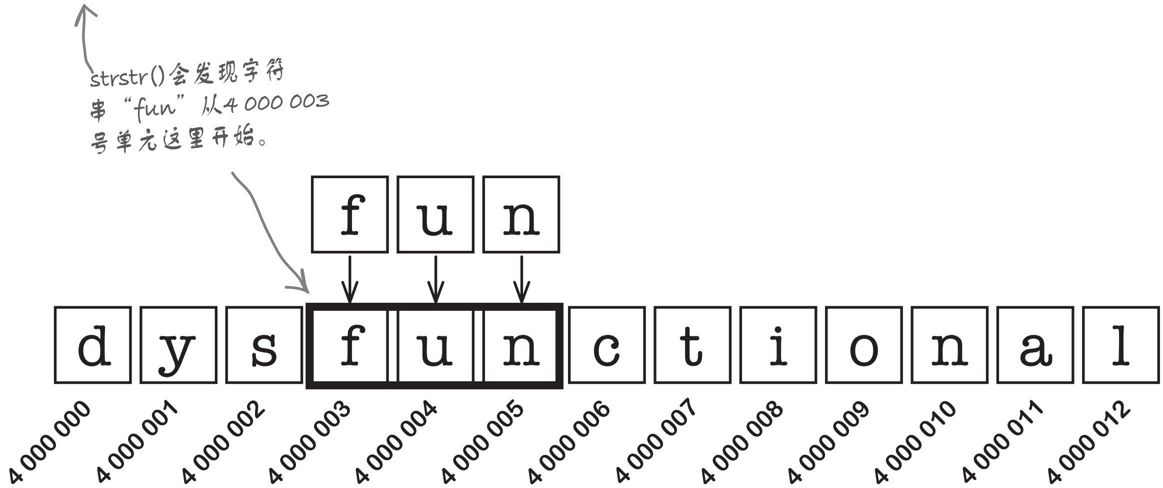
请写出点唱机程序要使用的那个函数。

strstr()

使用strstr()函数

为了解strstr()具体怎样工作，我们来看一个例子。假设要在一个长字符串“dysfunctional”中找字符串“fun”，你可以像这样调用strstr()：

```
strstr("dysfunctional", "fun")
```



strstr()函数会在第一个字符串中查找第二个字符串，如果找到，它会返回第二个字符串在存储器中的位置，在这个例子中，函数会发现fun子串从存储器4 000 003号单元开始。

如果strstr()找不到子串怎么办？如果找不到，strstr()就会返回0值。为什么要返回0呢？如果你还记得，就知道C语言中0就相当于假，也就是说，可以用strstr()检查一个字符串是否存在于另一个字符串中，像这样：

```
char s0[] = "dysfunctional";
char s1[] = "fun";
if (strstr(s0, s1))
  puts("我在dysfunctional中找到fun了!");
```

我们来看看如何在点唱程序中应用strstr()。

游泳池拼图

酒吧那帮家伙开始写点歌程序的代码了。哦，不！代码掉进了游泳池。你能选出正确的代码并补全“找歌”函数吗？游泳池已经几百年没人洗了，所以要小心，有的代码可能一次都用不到。

注意：这群家伙加入了一些他们在这本书中其他地方找到的代码。

void表示函数不会返回任何值。

嘿，瞧！你另外创建了一个函数。等你有时间写main()函数，就会调用这个函数。

```
void find_track(char search_for[])
{
  int i;
  for (i = 0; i < 5; i++) {
    if (..............(..............,..............))
      printf("Track %i: '%s'\n", .............., ..............);
  }
}
```

这是“for循环”，过会儿我们会深入学习，但现在只要知道它会将代码运行5次。

在这里检查搜索关键字是否出现在歌名中。

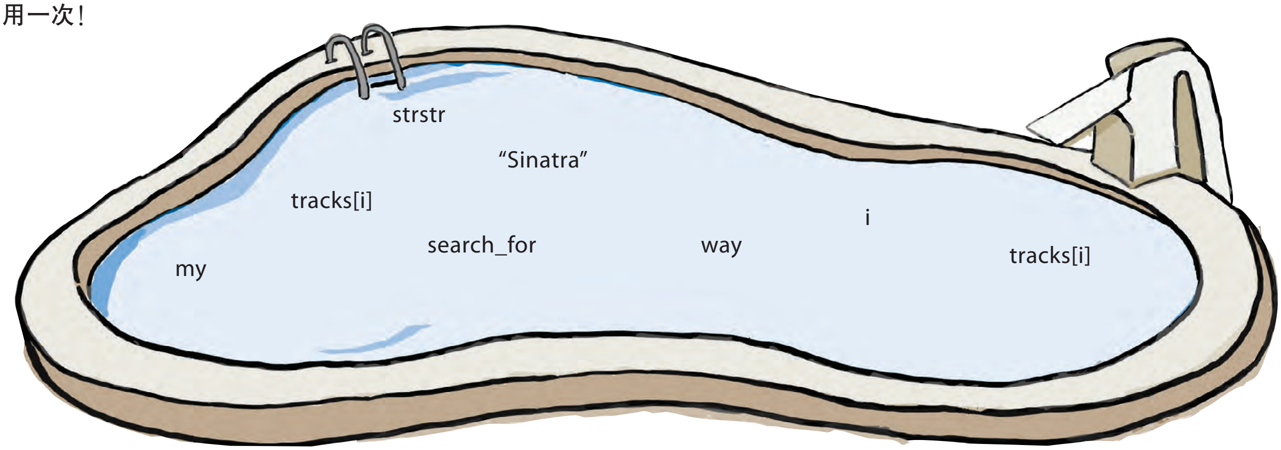
如果歌名和我们查找内容相匹配，就在这里显示歌名。

将在这里打印两个值。

一个是整型。

一个是字符串。

注意：每样东西只能使用一次！



变身编译器

点唱程序需要一个主函数，它从用户那里读取输入，然后调用左页的**find_track()**函数。你的任务是扮演编译器，说出你将在点唱程序中使用哪个**main()**函数。

```
int main()
{
  char search_for[80];
  printf("Search for: ");
  fgets(search_for, 80, stdin);
  find_track();
  return 0;
}
```

```
int main()
{
  char search_for[80];
  printf("Search for: ");
  fgets(search_for, 79, stdin);
  find_track(search_for);
  return 0;
}
```

```
int main()
{
  char search_for[80];
  printf("Search for: ");
  fgets(search_for, 80, stdin);
  search_for[strlen(search_for)-1]='\0';
  find_track(search_for);
  return 0;
}
```

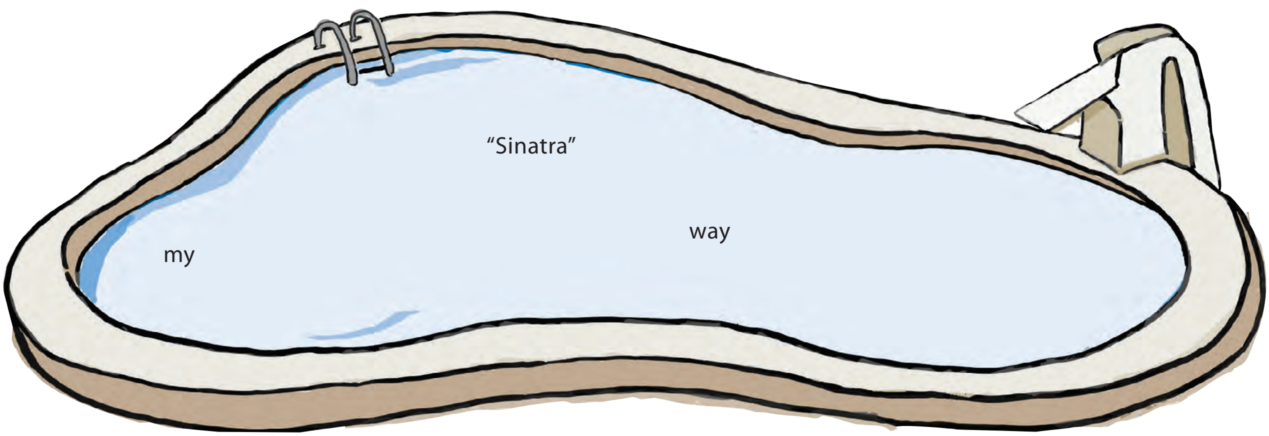
```
int main()
{
  char search_for[80];
  printf("Search for: ");
  scanf(search_for, 80, stdin);
  find_track(search_for);
  return 0;
}
```

游泳池拼图解答

酒吧那帮家伙开始写点歌程序的代码了。哦，不！代码掉进了游泳池。你将选出正确的代码并补全“找歌”函数。

注意：这群家伙加入了一些他们在这本书中其他地方找到的代码。

```
void find_track(char search_for[])
{
  int i;
  for (i = 0; i < 5; i++) {
    if ( strstr ( tracks[i] , search_for ))
      printf("Track %i: '%s'\n", i , tracks[i] );
  }
}
```



变身编译器解答

点唱程序需要一个主函数，它从用户那里读取输入，然后调用左页的**find_track()**函数。你的任务是扮演编译器，说出你将在点唱程序中使用哪个**main()**函数。

```
int main()
{
  char search_for[80];
  printf("Search for: ");
  fgets(search_for, 80, stdin);
  find_track();
  return 0;
}
```

调用find_track()时没有给它查询关键字。

```
int main()
{
  char search_for[80];
  printf("Search for: ");
  fgets(search_for, 79, stdin);
  find_track(search_for);
  return 0;
}
```

数组长度没有用全。写代码的人将数组的长度减1，你使用scanf()时才会这么做。

```
int main()
{
  char search_for[80];
  printf("Search for: ");
  fgets(search_for, 80, stdin);
  search_for[strlen(search_for)-1]='\0';
  find_track(search_for);
  return 0;
}
```

这是正确的main()函数。

```
int main()
{
  char search_for[80];
  printf("Search for: ");
  scanf(search_for, 80, stdin);
  find_track(search_for);
  return 0;
}
```

这次用了scanf()，但scanf()可不是这么写的。

该审查代码了

让我们把代码拼在一起，回顾一下到目前为止你做了哪些事：

还是需要stdio.h，因为要用printf()和scanf()函数。

还需要string.h头文件，因为要用strstr()函数来查找字符串。

把tracks数组放在main()和find_track()函数外，这样tracks就可以在程序的任何地方使用。

这是新find_track()函数，需要在main()中调用find_track()之前先声明它。

i++表示“i的值递增1”。

这段代码会显示所有匹配的歌曲。

这是main()函数，它是程序的起点。

在这里要求用户输入查找关键字。

现在调用新的find_track()函数，显示匹配的歌曲。

```
#include <stdio.h>
#include <string.h>

char tracks[][80] = {
  "I left my heart in Harvard Med School",
  "Newark, Newark - a wonderful town",
  "Dancing with a Dork",
  "From here to maternity",
  "The girl from Iwo Jima",
};

void find_track(char search_for[])
{
  int i;
  for (i = 0; i < 5; i++) {
    if (strstr(tracks[i], search_for))
      printf("Track %i: '%s'\n", i, tracks[i]);
  }
}

int main()
{
  char search_for[80];
  printf("Search for: ");
  fgets(search_for, 80, stdin);
  search_for[strlen(search_for)-1]='\0';
  find_track(search_for);
  return 0;
}
```

必须以这个顺序排列代码：在顶部包含头文件，这样编译器就能在编译代码前把所有函数都准备好。然后在开始写函数之前定义tracks，这叫把tracks数组放在全局域。全局变量位于任何函数之外，所有函数都可以调用它们。

最后，你有两个函数，find_track()在前，main()在后。find_track()必须赶在你在main()中调用它之前出现。

试驾

下面打开终端，看看代码能否工作。

```
File Edit Window Help string.h
> gcc text_search.c -o text_search && ./text_search
Search for: town
Track 1: 'Newark, Newark - a wonderful town'
>
```

好消息，程序工作了！

到目前为止，这个程序是你写过最长的一个，但它做了很多的事情。程序创建了一个字符串数组，并利用标准库中的字符串处理函数搜索数组中的歌名，最后找到了用户想要找的歌曲。

百宝箱

如果想了解更多关于*string.h*函数的信息，请参阅：http://tinyurl.com/82acwue。

如果你用的是Mac或Linux的计算机，可以在命令行中查看string.h中每个函数的详细介绍，假如想查看`strstr()`函数，可以输入：

```
man strstr
```

这里没有蠢问题

问： 为什么要把数组定义成`tracks[][80]`而不是`tracks[5][80]`？

答： 也可以这样定义，但编译器知道列表有5项，所以你可以省略5，写成[]。

问： 既然如此，为什么不直接写`tracks[][]`？

答： 每首歌的名字不一样长，为了放下最长的歌名，需要让编译器分配足够大的空间。

问： 也就是说`tracks`数组中每个字符串都有80个字符？

答： 程序会为每个字符串分配80个字符，即使歌名很短。

问： 所以`tracks`数组一共占了80×5=400字符？

答： 没错。

问： 如果我忘了包含*string.h*这样的头文件会怎么样？

答： 对于某些头文件来说，编译器会给出一个警告，但最后还是会包含它们；但对另一些来讲，编译器会直接提示编译错误。

问： 为什么我们要把`tracks`数组定义在函数外面？

答： 我们把`tracks`放在全局域，全局变量可以在所有函数中使用。

问： 既然我们创建了两个函数，计算机会先运行哪一个？

答： 程序总是首先运行`main()`函数。

问： 为什么我一定要把`find_track()`放在`main()`之前？

答： 在调用函数前，编译器需要知道两件事，函数接收什么参数以及函数的返回类型是什么。

问： 如果我把`main()`放到前面会怎么样？

答： 你会得到几个警告。

要点

- 可以用`char strings[...][...]`来创建数组的数组。
- 第一组方括号用来访问外层数组。
- 第二组方括号用来访问每个内层数组中的元素。
- 有了*string.h*头文件，就可以使用C标准库中的字符串处理函数。
- 可以在一个C程序中创建多个函数，但计算机总是先运行`main()`。

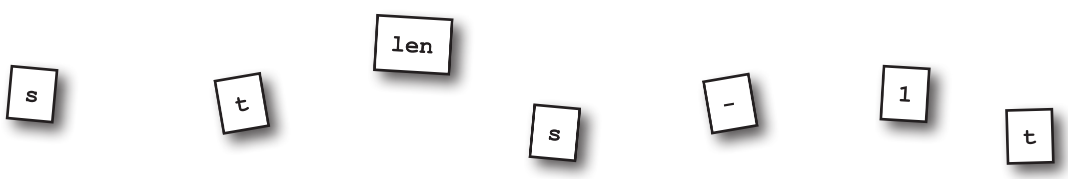
代码冰箱贴

这群人正在为一款小游戏编写新的代码，他们创建了一个函数，可以倒过来显示字符串。但悲剧发生了，一部分冰箱贴乱了，你能帮助他们重组代码吗？

size_t相当于整型，用来保存字符串的长度。

计算出字符串的长度，strlen("ABC")==3。

```
void print_reverse(char *s)
{
  size_t len = strlen(s);

  char *t = .......... + .......... - 1;

  while ( .......... >= .......... ) {
    printf("%c", *t);

    t = ..........  ..........  ..........;
  }
  puts("");
}
```



代码冰箱贴解答

一群人正在为一款小游戏编写新的代码。他们创建了一个函数，可以倒过来显示字符串。但悲剧发生了，一部分冰箱贴乱了，还好有你帮助他们重组代码。

```
void print_reverse(char *s)
{
  size_t len = strlen(s);

  char *t = s + len - 1;

  while ( t >= s ) {
    printf("%c", *t);

    t = t - 1;
  }
  puts("");
}
```

像这样计算地址就叫“指针算术运算”。

“数组的数组”和“指针的数组”

你已经见过了如何使用数组的数组保存多个字符串，还有一种方法是使用指针的数组。顾名思义，指针的数组就是保存存储器地址的数组。如果想要快速创建字符串字面值列表，指针的数组就非常有用：

```
char *names_for_dog[] = {"Bowser", "Bonza", "Snodgrass"};
```

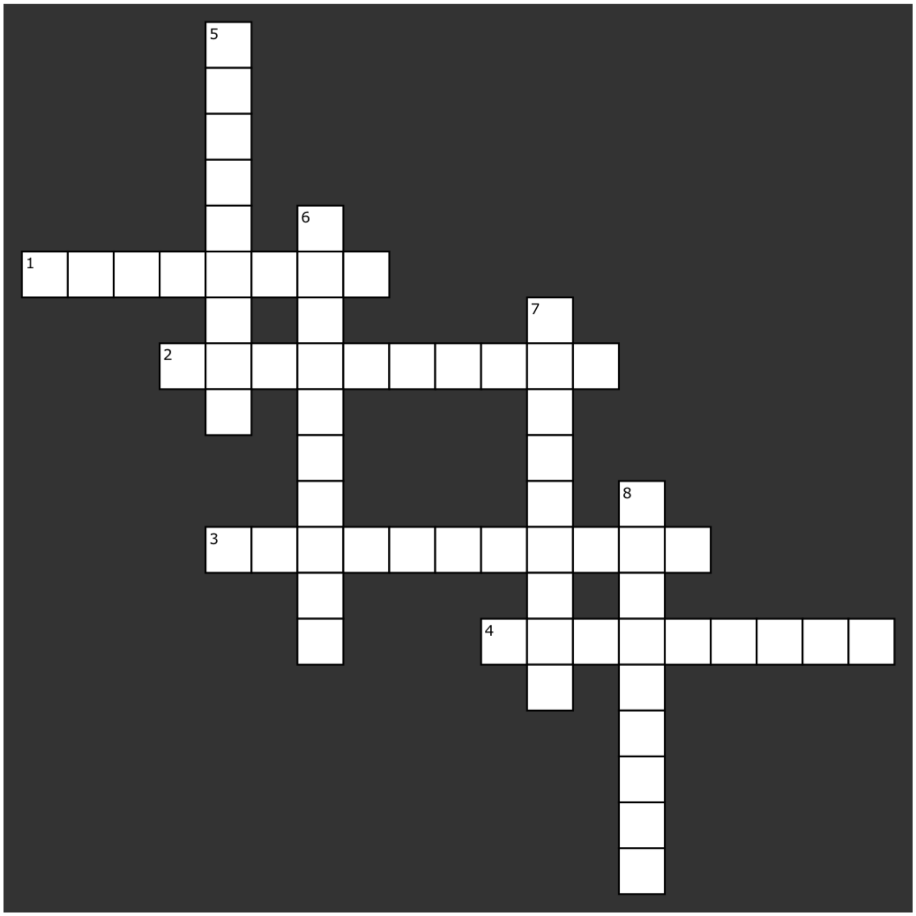
保存指针的数组。

一个字符串字面值配一个指针。

你可以像访问数组的数组那样访问指针的数组。

填字游戏

这群人用**print_reverse()**写了一个填字游戏，程序的输出就是填字游戏的答案。



横

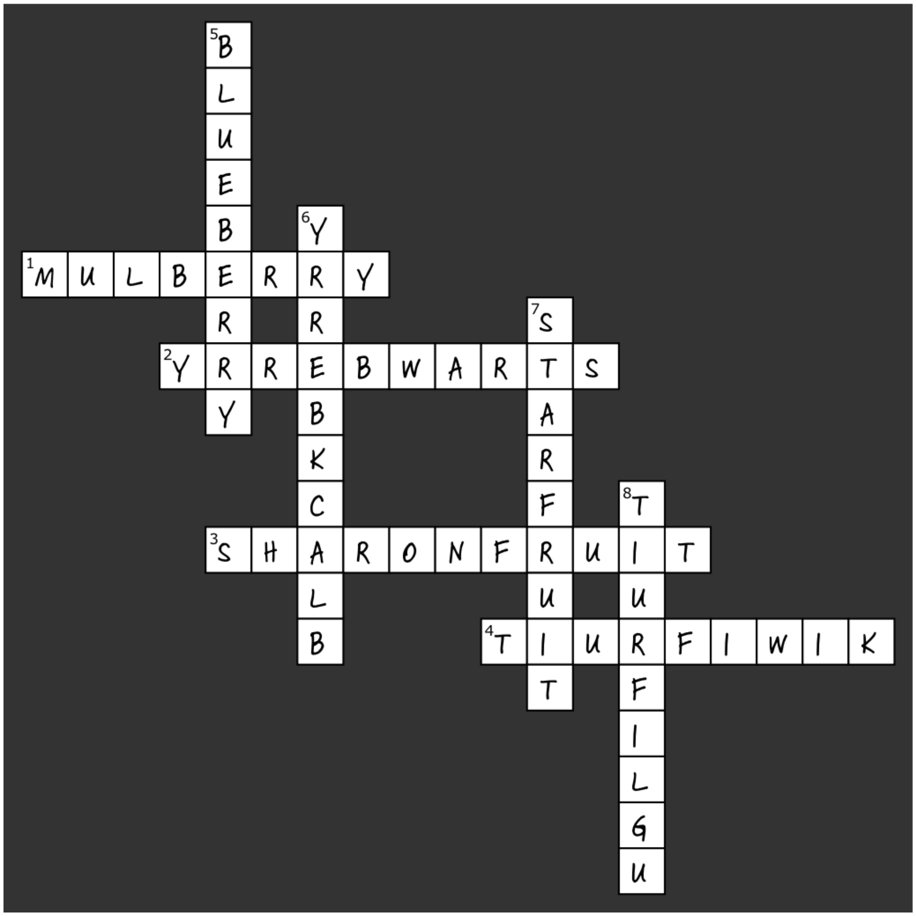
```
   int main()
   {
     char *juices[] = {
       "dragonfruit", "waterberry", "sharonfruit", "uglifruit",
       "rumberry", "kiwifruit", "mulberry", "strawberry",
       "blueberry", "blackberry", "starfruit"
     };
     char *a;
1    puts(juices[6]);
2    print_reverse(juices[7]);
     a = juices[2];
     juices[2] = juices[8];
     juices[8] = a;
3    puts(juices[8]);
4    print_reverse(juices[(18 + 7) / 5]);
```

纵

```
5    puts(juices[2]);
6    print_reverse(juices[9]);
     juices[1] = juices[3];
7    puts(juices[10]);
8    print_reverse(juices[1]);
     return 0;
   }
```

填字游戏解答

这群人用**print_reverse()**写了一个填字游戏，程序的输出就是填字游戏的答案。



横

```
int main()
{
  char *juices[] = {
    "dragonfruit", "waterberry", "sharonfruit", "uglifruit",
    "rumberry", "kiwifruit", "mulberry", "strawberry",
    "blueberry", "blackberry", "starfruit"
  };
  char *a;
1 puts(juices[6]);
2 print_reverse(juices[7]);
  a = juices[2];
  juices[2] = juices[8];
  juices[8] = a;
3 puts(juices[8]);
4 print_reverse(juices[(18 + 7) / 5]);
```

纵

```
5 puts(juices[2]);
6 print_reverse(juices[9]);
  juices[1] = juices[3];
7 puts(juices[10]);
8 print_reverse(juices[1]);
  return 0;
}
```

C语言工具箱

学完第2.5章，现在你的工具箱中又多了字符串。关于本书提示工具条的完整列表，请见附录ii。

字符串数组是数组的数组。

string.h头文件包含了有用的字符串处理函数。

可以用char strings[...][...]创建数组的数组。

strcmp()可以比较字符串。

strstr(a, b)可以返回字符串b在字符串a中的地址。

strchr()用来在字符串中找到某个字符的位置。

strcat()可以连接字符串

strcpy()可以复制字符串。

strlen()可以得到字符串的长度。

3 创建小工具

做一件事并把它做好

操作系统都有小工具。

C语言小工具执行特定的小任务，例如读写文件、过滤数据。如果想要完成更复杂的任务，可以把多个工具链接在一起。那么如何构建小工具呢？本章中，你会看到创建小工具的基本要素并学会控制命令行选项、操纵信息流、重定向，并很快建立自己的工具。

小工具可以解决大问题

小工具是一个C程序，它做一件事情并把它做好。小工具可以做各种事情，例如在屏幕上显示文件的内容，列出计算机上正在运行的进程，显示文件前10行的内容，或把这些内容发送到打印机。有操作系统的地方就有小工具，可以在命令提示符或终端运行这些工具。当你想解决一个大问题时，可以把它分解成一连串的小问题，然后针对每个小问题写一个小工具。

通常，像Linux这样的操作系统由无数小工具组成。

小工具做一件事，并把它做好。

有人为我写了一个地图应用程序，我想利用它发布行车路线，但GPS输出数据的格式对不上。

这是自行车手的GPS输出的数据，它们用逗号分隔。

这是纬度。

这是经度。

```
42.363400,-71.098465,Speed = 21
42.363327,-71.097588,Speed = 23
42.363255,-71.096710,Speed = 17
```

这是地图应用需要的格式，用JavaScript对象表示法(JavaScript Object Notation)表示，即JSON。

数据一样，但格式不同。

```
data=[
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
...
]
```

程序的某个模块需要转化数据的格式，这样的任务用小工具来完成再适合不过了。

袖珍代码

嘿，我们都写过这样的代码，因为使用了太多打印语句，导致代码难以阅读。但只要细心，一定能拼凑出原来的代码。

下面这个程序从命令行读取用逗号分隔的数据，然后以JSON格式显示，你能补上漏掉的代码吗？

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  int started = ................;

  puts("data=[");
  while (scanf("%f,%f,%79[^\n]", ................, ................, ................) == 3) {
    if (started)
      printf(",\n");
    else
      started = ................;
    printf("{latitude: %f, longitude: %f, info: '%s'}",............, ..............., ........);
  }
  puts("\n]");
  return 0;
}
```

我们用scanf()输入多条数据。

这里填什么？别忘了，scanf()总是接收指针参数。

scanf()函数返回成功读取的数据条数。

相当于在说：“把这一行余下来的字符都给我。”

设置started时要小心。

需要显示什么值？

袖珍代码解答

嘿，我们都写过这样的代码，因为使用了太多打印语句，导致代码难以阅读。但细心的你一定拼凑出了原来的代码。

下面这个程序从命令行读取用逗号分隔的数据，然后以JSON格式显示，你将补上漏掉的代码。

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  int started = 0;

  puts("data=[");
  while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3) {
    if (started)
      printf(",\n");
    else
      started = 1;
    printf("{latitude: %f, longitude: %f, info: '%s'}", latitude, longitude, info);
  }
  puts("\n]");
  return 0;
}
```

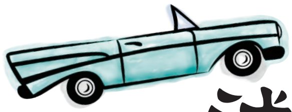
我们需要把started的初值设为0，表示假。

还记得数值变量前的&代表什么吗？scanf()接收指针。

如果已经显示过了一行数据，就用逗号把它隔开。

循环执行一遍以后，就可以把started设为1，表示真。

这里不需要&，因为printf()接收的是值，而不是数值的地址。



试驾

当你编译并运行这段代码时会发生什么？程序会做哪些事情？

这是程序打印的数据。

这是输入的数据。

输入和输出混在了一起。

```
File Edit Window Help JSON
>./geo2json
data=[
42.363400,-71.098465,Speed = 21
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'}42.363327,-71.097588,Speed = 23
,
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'}42.363255,-71.096710,Speed = 17
,
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'}42.363182,-71.095833,Speed = 22
,
...
...
...
{latitude: 42.363182, longitude: -71.095833, info: 'Speed = 22'}42.362385,-71.086182,Speed = 21
,
{latitude: 42.362385, longitude: -71.086182, info: 'Speed = 21'}^D
]
>
```

艰苦奋战了几个小时以后……

最后，需要按Ctrl-D停止程序。

程序要你在键盘输入GPS数据，然后它在屏幕上显示JSON格式的数据，于是输入和输出数据混作一团，而且数据量还很大。如果你要写一个小工具，一定不想手工输入数据，而是希望从文件中读取大量数据。

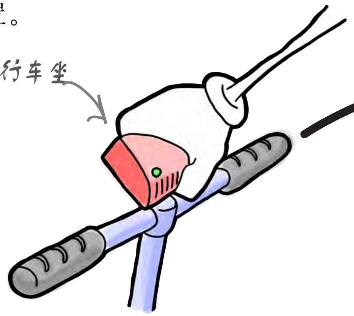
如何使用JSON数据也是个问题，打印在屏幕上肯定不管用。

那么程序正确运行了吗？它把事情做好了吗？**需要修改代码吗？**

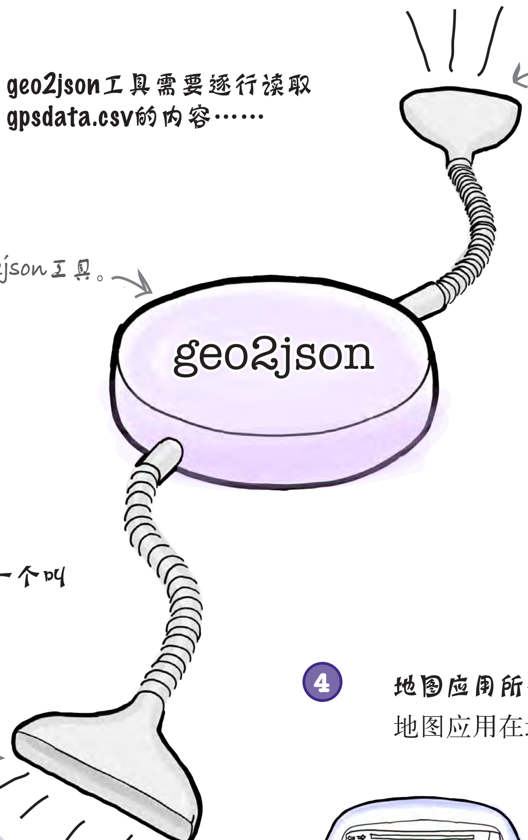
程序如何工作

1 从自行车上取下GPS，下载数据。

GPS会创建一个叫*gpsdata.csv*的文件，文件中每一行数据都代表一个位置。



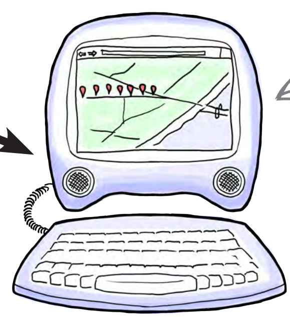
2 geo2json工具需要逐行读取gpsdata.csv的内容……



3 然后以JSON格式把数据写到一个叫output.json的文件中。

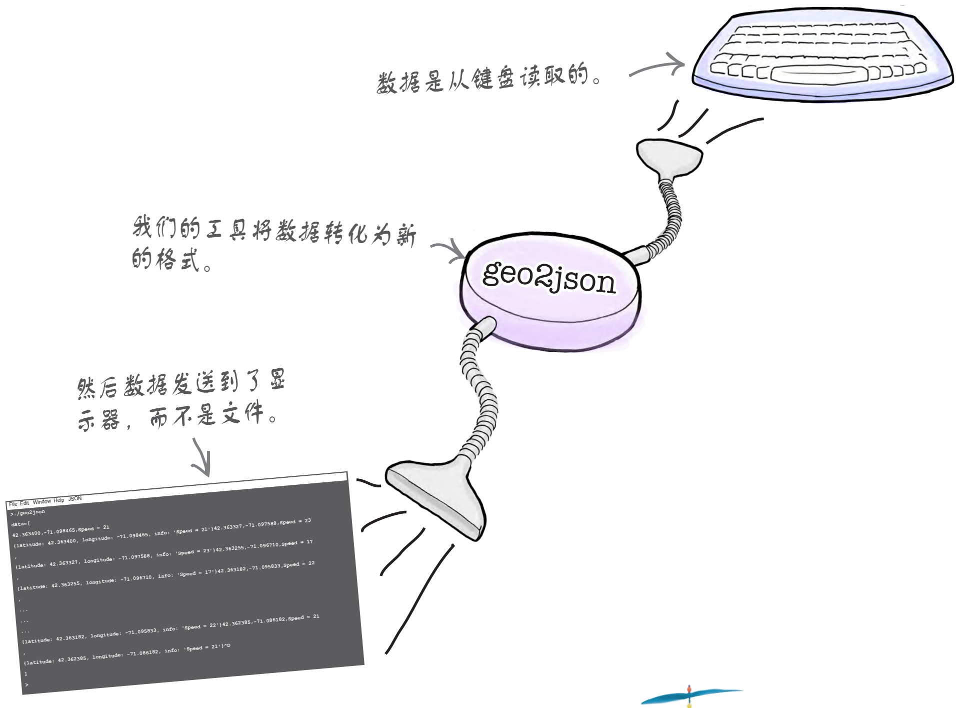
4 地图应用所在的网页读取output.json文件。

地图应用在地图上显示所有坐标。



但没有使用文件……

目前程序读写的对象不是文件，而是键盘和显示器。



这样做还不够好，既然数据已经保存在了文件中，用户可不想再输入一遍。况且在屏幕上显示JSON格式的数据，网页中的地图也读不到。

想让程序使用文件，该怎么做？如果你想用文件代替键盘与显示器，需要修改哪些代码？非要修改代码不可吗？

脑力风暴

有没有什么办法不用改代码，甚至不用重新编译，就能让程序使用文件？

百宝箱

有一种小工具叫过滤器（filter），它逐行读取数据，对数据进行处理，再把数据写到某个地方。如果你的计算机是Unix，或你在Windows上安装了Cygwin，就已经拥有很多过滤器工具了。

head：显示文件前几行的内容。

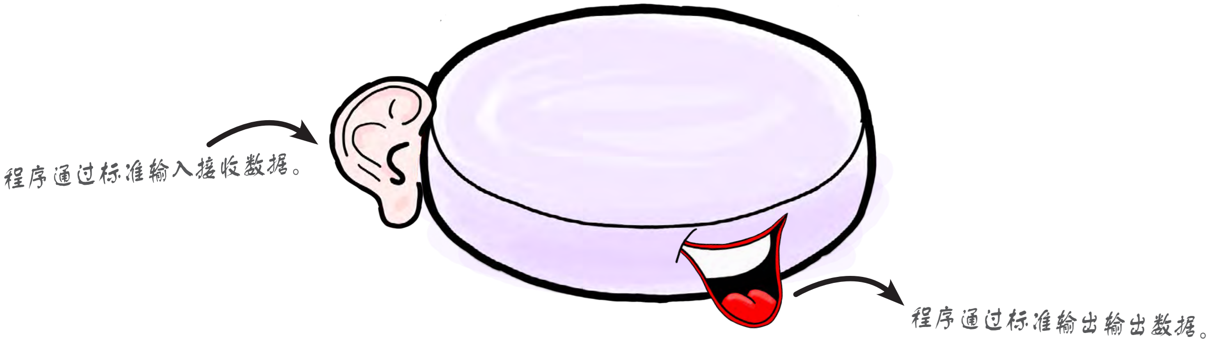
tail：显示文件最后几行的内容。

sed：流编辑器（stream editor），用来搜索和替换文本。

等会儿你会看到如何把多个过滤器组合在一起，形成过滤器链。

可以用重定向

在用scanf()从键盘读取数据、printf()向显示器写数据时，这两个函数其实并没有直接使用键盘、显示器，而是用了标准输入和标准输出。程序运行时，操作系统会创建标准输入和标准输出。



操作系统控制数据如何进出标准输入、标准输出。如果在命令提示符或终端运行程序，操作系统会把所有键盘输入都发送到标准输入；默认情况下，如果操作系统从标准输出中读到数据，就发送到显示器。

scanf()和printf()函数并不知道数据从哪里来，也不知道数据要到哪里去，它们也不关心这点，它们只管从标准输入读数据，向标准输出写数据。

听起来有些故弄玄虚，为什么不让程序直接使用键盘和屏幕呢？岂不是更简单？

操作系统为什么要使用标准输入、标准输出与程序交互呢？有一个很好的原因：

因为这么一来，就可以重定向标准输入、标准输出，让程序从键盘以外的地方读数据、往显示器以外的地方写数据，例如文件。

可以用 < 重定向标准输入……

你不必再用键盘输入数据，可以使用 < 操作符从文件中读取数据。

```
42.363400,-71.098465,Speed = 21
42.363327,-71.097588,Speed = 23
42.363255,-71.096710,Speed = 17
42.363182,-71.095833,Speed = 22
42.363110,-71.094955,Speed = 14
42.363037,-71.094078,Speed = 16
42.362965,-71.093201,Speed = 18
42.362892,-71.092323,Speed = 22
42.362820,-71.091446,Speed = 17
42.362747,-71.090569,Speed = 23
42.362675,-71.089691,Speed = 14
42.362602,-71.088814,Speed = 19
42.362530,-71.087936,Speed = 16
42.362457,-71.087059,Speed = 16
42.362385,-71.086182,Speed = 21
```

这个文件含有GPS设备输出的数据。

让操作系统把数据从文件发送到程序的标准输入。

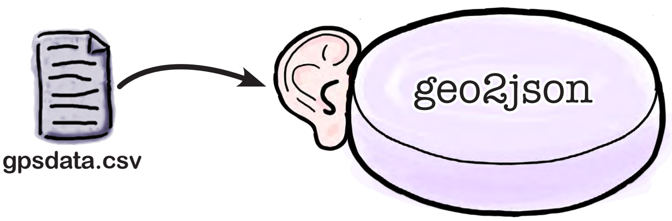
不必输入GPS数据，它们也就不会和输出混在一起。

File Edit Window Help Don'tCrossTheStreams

```
> ./geo2json < gpsdata.csv
data=[
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
{latitude: 42.363182, longitude: -71.095833, info: 'Speed = 22'},
{latitude: 42.363110, longitude: -71.094955, info: 'Speed = 14'},
{latitude: 42.363037, longitude: -71.094078, info: 'Speed = 16'},
...
...
{latitude: 42.362385, longitude: -71.086182, info: 'Speed = 21'}
]
>
```

现在只看到了程序输出的JSON数据。

<操作符告诉操作系统，程序的标准输入应该与*gpsdata.csv*文件相连，而不是键盘，所以可以把数据从文件发送到程序。现在只需重定向程序的输出。



用 > 重定向标准输出

为了把标准输出重定向到文件，需要使用 > 操作符：

现在，同时重定向了标准输入和标准输出。

```
File Edit Window Help Don'tCrossTheStreams
> ./geo2json < gpsdata.csv > output.json
>
```

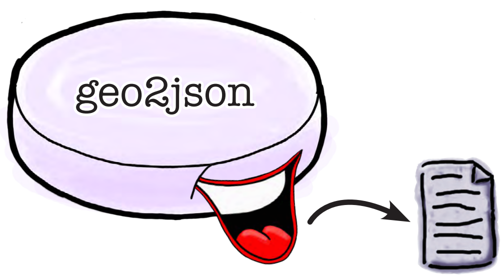
程序的输出现在写进了output.json。

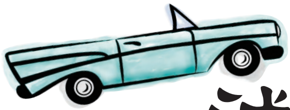
```
data=[
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
{latitude: 42.363182, longitude: -71.095833, info: 'Speed = 22'},
{latitude: 42.363110, longitude: -71.094955, info: 'Speed = 14'},
{latitude: 42.363037, longitude: -71.094078, info: 'Speed = 16'},
{latitude: 42.362965, longitude: -71.093201, info: 'Speed = 18'},
{latitude: 42.362892, longitude: -71.092323, info: 'Speed = 22'},
{latitude: 42.362820, longitude: -71.091446, info: 'Speed = 17'},
{latitude: 42.362747, longitude: -71.090569, info: 'Speed = 23'},
{latitude: 42.362675, longitude: -71.089691, info: 'Speed = 14'},
{latitude: 42.362602, longitude: -71.088814, info: 'Speed = 19'},
{latitude: 42.362530, longitude: -71.087936, info: 'Speed = 16'},
{latitude: 42.362457, longitude: -71.087059, info: 'Speed = 16'},
{latitude: 42.362385, longitude: -71.086182, info: 'Speed = 21'}
]
```

显示器上没有任何输出，因为数据都写到了output.json文件中。

因为重定向了标准输出，所以屏幕上没有出现任何数据，程序现在创建了一个叫*output.json*的文件。

地图应用需要用到*output.json*文件，让我们看看地图能否工作。





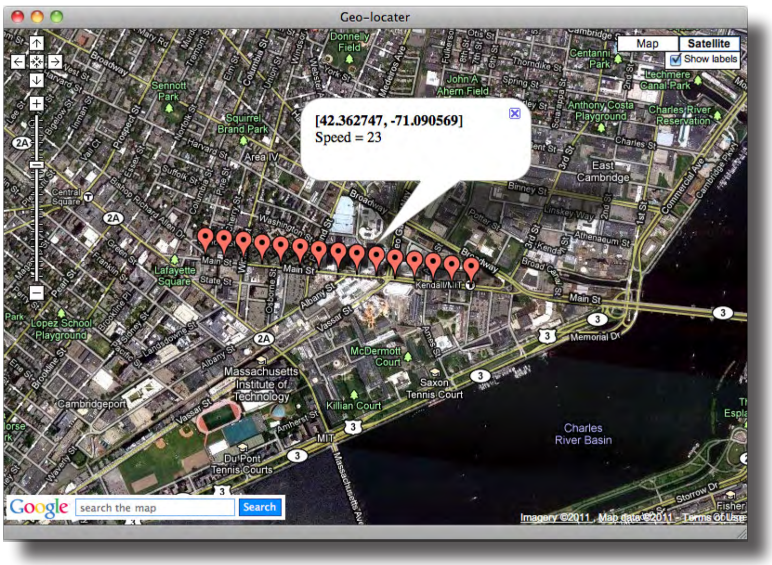
试驾

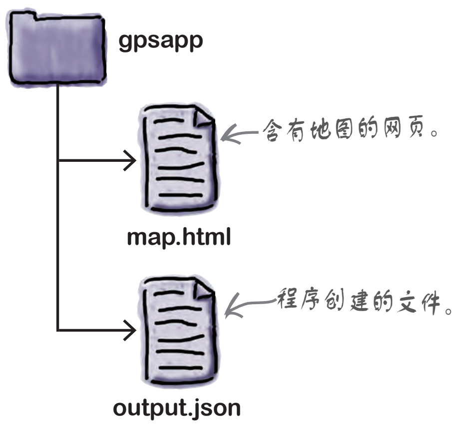
试试新创建的数据文件能否在地图上画出坐标，把含有地图程序的网页复制到*output.json*所在的文件夹中，然后用浏览器打开网页。

动起来！

从以下地址下载网页：

http://chengyichao.info/hfc/ditu.html





地图工作了。

网页中的地图成功读取了输出文件中的数据。

一些数据出错了……

程序已经能够顺利读取GPS数据，并把数据转化为地图应用需要的格式。但是几天以后，程序出现了一个问题。

发生了什么事情？原来GPS数据文件中有一些错误数据：

```
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
{latitude: 423.63182, longitude: -71.095833, info: 'Speed = 22'},
```

小数点出现在了错误的位置。

`geo2json`程序并不会检查读入的数据，它只是改变数字的格式，然后把它们发送到输出文件。

这个问题应该不难解决，你需要校验数据。

你需要在geo2json程序中添加一些代码，用来检查错误的经、纬度值。无需任何高深的技巧。假如经度或纬度不在指定的范围内，就显示一条错误消息，并在退出程序的同时把错误状态码置为2：

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  int started = 0;

  puts("data=[");
  while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3) {
    if (started)
      printf(",\n");
    else
      started = 1;
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
..................................................................
    printf("{latitude: %f, longitude: %f, info: '%s'}", latitude, longitude, info);
  }
  puts("\n]");
  return 0;
}
```

纬度小于-90或大于90，退出程序并把错误状态码置为2；经度小于-180或大于180，退出程序并把错误状态码置为2。

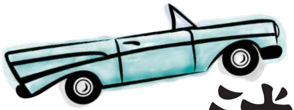
你在geo2json程序中添加一些代码，用来检查错误的经、纬度值。假如经度或纬度不在指定的范围内，就显示一条错误消息，并在退出程序的同时把错误状态码置为2：

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  int started = 0;

  puts("data=[");
  while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3) {
    if (started)
      printf(",\n");
    else
      started = 1;
    if ((latitude < -90.0) || (latitude > 90.0)) {
      printf("Invalid latitude: %f\n", latitude);
      return 2;
    }
    if ((longitude < -180.0) || (longitude > 180.0)) {
      printf("Invalid longitude: %f\n", longitude);
      return 2;
    }

    printf("{latitude: %f, longitude: %f, info: '%s'}", latitude, longitude, info);
  }
  puts("\n]");
  return 0;
}
```

判断经度与纬度的值是否在正确的范围内。

程序从main()函数中退出，并将错误状态码置为2。

显示简单的错误消息。



试驾

好啦，代码现在可以检查经度、纬度的范围了，程序能发现错误数据吗？我们拭目以待。

编译代码，使用错误数据作输入，运行程序：

重新编译程序。

然后用错误的数据再运行一次程序。

TMD！没有错误消息？意思是“挺美的”。

把输出保存到output.json文件。

```
File Edit Window Help Don'tCrossTheStreams
> gcc geo2json.c -o geo2json
> ./geo2json < gpsdata.csv > output.json
>
```

坐标哪去了？

太奇怪了，明明加了检查错误的代码，但运行程序时，一切还是老样子，但这次地图上一个坐标都没有，这是为什么？

脑力风暴

好好研究一下这段代码，你认为发生了什么？代码有没有照你说的去做？为什么连一条错误消息都没有？为什么地图应用认为整个*output.json*文件是错的？

代码拆析

既然地图程序在埋怨*output.json*文件，那我们就打开它，看看里面是什么：

这是output.json文件。

```
data=[
{latitude: 42.363400, longitude: -71.098465, info: 'Speed = 21'},
{latitude: 42.363327, longitude: -71.097588, info: 'Speed = 23'},
{latitude: 42.363255, longitude: -71.096710, info: 'Speed = 17'},
Invalid latitude: 423.631805
```

原来错误消息也被重定向到了输出文件中。

一打开文件，你就可以看得一清二楚。当程序一看到错误数据就马上退出了，它不再继续处理数据，而是**输出**了一条错误消息。当把标准输出重定向到*output.json*，也重定向了错误消息，于是程序一声不吭地结束，你永远不知道问题出在哪里。

如果你看得到错误消息，就会去检查程序的退出状态，但你现在连程序出错了都不知道。

怎样才能在重定向输出的同时显示错误消息呢?

百宝箱

程序在数据中发现错误就会退出，并把退出状态置为2。怎么在程序结束后检查错误状态呢？要看操作系统，如果你的计算机是Mac、Linux、其他UNIX，或你在Windows上使用Cygwin，可以用以下命令显示错误状态：

```
File Edit Window Help
$ echo $?
2
```

如果用的是Windows的命令提示符，则可以输入：

```
File Edit Window Help
C:\> echo %ERRORLEVEL%
2
```

这两条命令做了相同的事：显示程序结束时返回的那个数字。

要是有一种针对错误的输出就好了，这样我就不会把错误消息混到标准输出里了，但我知道这是在白日做梦……

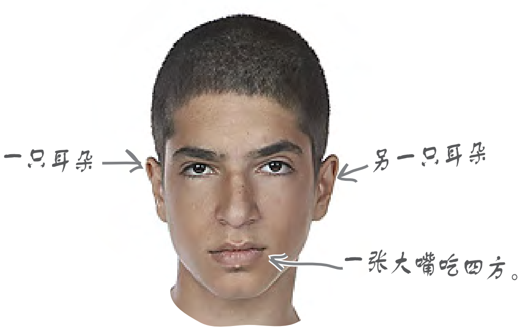
隆重推出标准错误

标准输出是程序输出数据的默认方式。但如果发生了意外该怎么办？比如出错了。你一定想区分错误消息和普通输出。

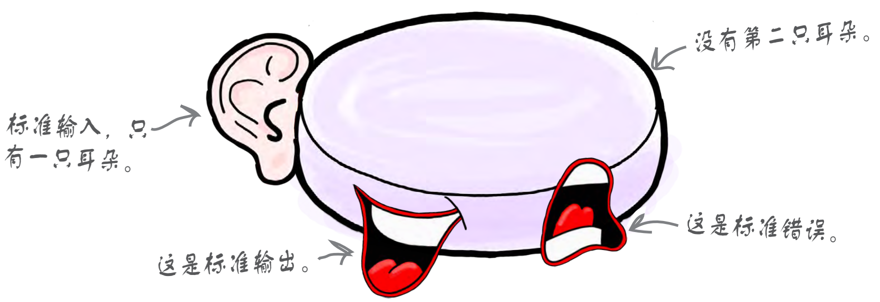
这就是为什么要发明**标准错误**——一个用来发送错误消息的二号输出。

人有两只耳朵和一张嘴，但进程有一只耳朵（标准输入）和两张嘴（标准输出和标准错误）。

人



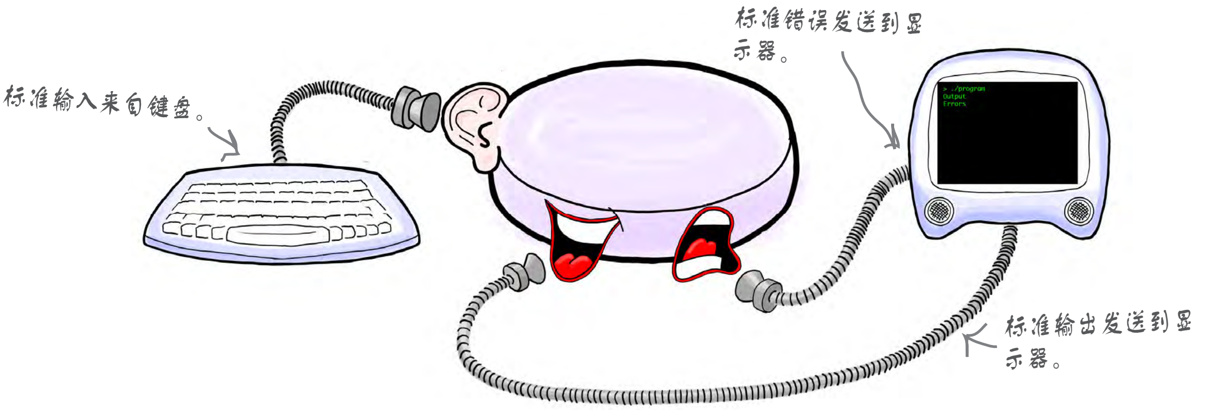
进程



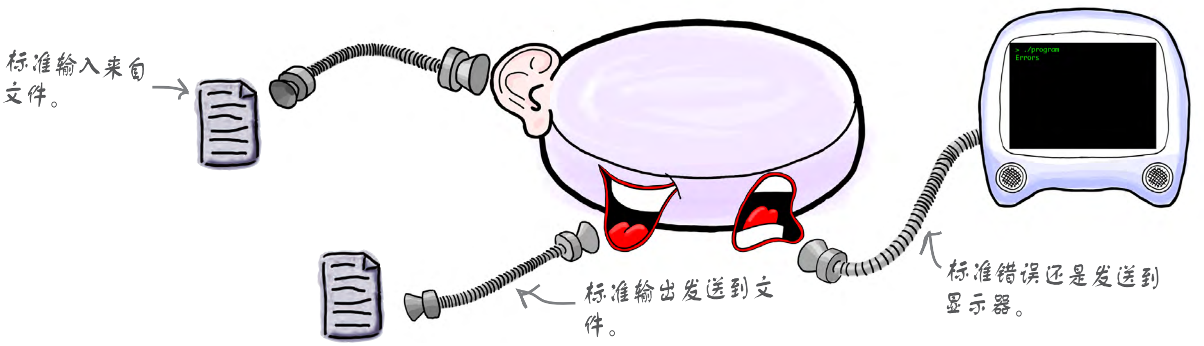
看看操作系统如何建立耳朵和嘴巴。

默认情况下，标准错误会发送到显示器

还记得吗？当操作系统创建了一个新的进程，它会把标准输入指向键盘，而将标准输出指向屏幕。同时操作系统还会创建标准错误，和标准输出一样，标准错误会默认发送到显示器。



这意味着当某人把标准输入和标准输出重定向到了文件，标准错误仍然会把数据发送到显示器。



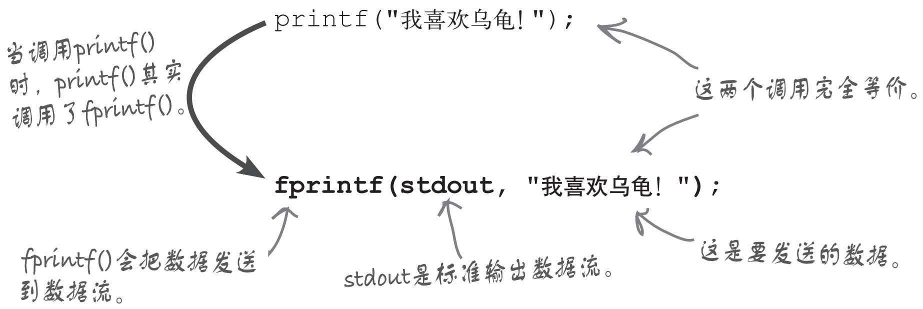
太好了！也就是说即使把标准输出重定向到了其他地方，默认情况下，所有发送到标准错误的消息依然会显示在屏幕上。

只要用标准错误显示错误消息，问题就迎刃而解。

具体怎么做？

fprintf()打印到数据流

printf()函数可以将数据发送到标准输出，但printf()其实只是一个函数的特例，而这个函数叫fprintf()。



fprintf()函数可以让你决定把文本发送到哪里，你既可以让fprintf()把文本发送到**stdout**（标准输出），也可以发送到**stderr**（标准错误）。

这里没有蠢问题

问：既然有stdout和stderr，自然就有stdin吧？

答：有，如你所料，它代表标准输入。

问：我可以打印stdin吗？

答：不可以。

问：我可以从stdin中读取数据吗？

答：嗯，你可以用fscanf()来读取，它的用法和scanf()很像，区别是可以指定fscanf()从哪条数据流中读取数据。

问：也就是说fscanf(stdin,...)和scanf()等价？

答：没错，它们完全相同。说到底，scanf()就是用fscanf(stdin, ...)实现的。

问：我可以重定向标准错误吗？

答：可以，你可以用>重定向标准输出，2>重定向标准错误。

问：所以我要写geo2json 2> errors.txt?

答：没错。

用fprintf()修改代码吧

只要稍作修改，就可以在标准错误中打印错误消息。

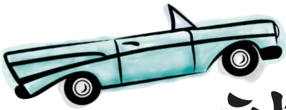
```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  int started = 0;

  puts("data=[");
  while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3) {
    if (started)
      printf(",\n");
    else
      started = 1;
    if ((latitude < -90.0) || (latitude > 90.0)) {
      printf("Invalid latitude: %f\n", latitude);
      fprintf(stderr, "Invalid latitude: %f\n", latitude);
      return 2;
    }
    if ((longitude < -180.0) || (longitude > 180.0)) {
      printf(stderr, "Invalid longitude: %f\n", longitude);
      fprintf(stderr, "Invalid longitude: %f\n", longitude);
      return 2;
    }
    printf("{latitude: %f, longitude: %f, info: '%s'}", latitude, longitude, info);
  }
  puts("\n]");
  return 0;
}
```

用fprintf()代替printf()。

我们需要把第一个参数设为stderr。

现在程序和刚刚一样工作，只是错误消息会显示在标准错误中，而非标准输出中。

运行代码，看看会发生什么。



试驾

重新编译程序，再次运行错误的GPS数据，结果如下：

```
File Edit Window Help ControlErrors
> gcc geo2json.c -o geo2json
> ./geo2json  < gpsdata.csv > output.json
Invalid latitude: 423.631805
```

妙哉，这次就算把标准输出重定向到*output.json*文件，也可以在屏幕上看到错误消息。

创建标准错误的初衷是为了区分普通输出和错误消息。但是别忘了，stderr和stdout不过是两个输出流罢了，完全可以用它们做其他事情。

下面就来试试你新学的两个技能：标准输入和标准错误。

要点

- printf()函数把数据发送到标准输出。
- 默认情况下，标准输出会发送到显示器。
- 可以在命令行中用>将标准输出重定向到文件。
- scanf()从标准输入读取数据。
- 默认情况下，标准输入会从键盘读取数据。
- 可以在命令行中用<将标准输入重定向到文件。
- 标准错误专门用来输出错误消息。
- 可以用2>重定向标准错误。

最高机密

毫无疑问，下面这个程序可以用来传送机密消息：

```
#include <stdio.h>

int main()
{
  char word[10];
  int i = 0;
  while (scanf("%9s", word) == 1) {
    i = i + 1;
    if (i % 2)
      fprintf(stdout, "%s\n", word);
    else
      fprintf(stderr, "%s\n", word);
  }
  return 0;
}
```

i%2表示"i除以2的余数"。

我们截获了一个叫*secret.txt*的文件，还有一张小纸片，上面写着指令：

```
THE BUY SUBMARINE
SIX WILL EGGS
SURFACE AND AT
SOME NINE MILK PM
```

secret.txt

运行以下命令：

```
secret_messages < secret.txt > message1.txt 2> message2.txt
```

>会重定向标准输出。

2>会重定向标准错误。

你负责解码这两条机密消息，请把答案写在下面。

消息1	**消息2**
....	....
....	....
....	....
....	....
....	....
....	....
....	....

最高机密解答

毫无疑问，下面这个程序可以用来传送机密消息：

```
#include <stdio.h>

int main()
{
  char word[10];
  int i = 0;
  while (scanf("%9s", word) == 1) {
    i = i + 1;
    if (i % 2)
      fprintf(stdout, "%s\n", word);
    else
      fprintf(stderr, "%s\n", word);
  }
  return 0;
}
```

我们截获了一个叫*secret.txt*的文件，还有一张小纸片，上面写着指令：

```
THE BUY SUBMARINE
SIX WILL EGGS
SURFACE AND AT
SOME NINE MILK PM
```

secret.txt

运行以下命令：

`secret_messages < secret.txt > message1.txt 2> message2.txt`

你负责解码这两条机密消息。

消息 1	消息 2
THE	BUY
SUBMARINE	SIX
WILL	EGGS
SURFACE	AND
AT	SOME
NINE	MILK
PM	

操作系统零距离

本周访谈主题：一视同仁

Head First: 操作系统，很高兴你抽空参加我们今天的节目。

O/S: 分配时间是我的强项。

Head First: 你不打算透露你的真实姓名，对吗？

O/S:是的，叫我O/S就行了。

Head First: 你属于哪一类操作系统？这个问题你介意回答吗？

O/S:大家总是在争论哪个操作系统好，但对C程序来说，其实我们都差不多。

Head First: 是因为有C语言标准库的缘故吗？

O/S: 嗯，C语言基本原理是放之四海皆准的。我常说“只要灯一关，我们都一个样”，你明白我在说什么吗？

Head First: 当然。现在是你负责把程序载入存储器的吗？

O/S: 没错，我把程序变成进程。

Head First: 这是很重要的工作，对吗？

O/S:当然啦，你可不能把程序扔到存储器中，让它自生自灭，还有一大堆配置工作要做。我需要为程序分配存储器，把程序和标准数据流连到一起，这样程序才能使用显示器和键盘。

Head First: 就像你对`geo2json`做的那样吗？

O/S: 它很傻。

Head First: 傻？

O/S: 不是说它真傻，而是作为一个工具，它操作起来很简单，好比一台傻瓜相机。

Head First: 原来是这样，你会用很多工具吗？

O/S: 这其实就是生活，不是吗？要看操作系统，类Unix的操作系统为完成工作会大量使用工具，Windows用的少一些，但也不能没有这些小工具。

Head First: 可以创建很多小工具并让它们在一起工作，这是一种哲学，对吗？

O/S: 这是一种生活方式。有时当你要解决一个大问题，把它分解成一组更简单的任务，解决起来更容易。

Head First: 然后为每个任务都写一个工具？

O/S: 没错，然后操作系统，也就是我，负责把这些工具连接起来。

Head First: 这种方法有什么好处？

O/S: 首先是简单，小程序更容易测试。其次，一旦你写了一个工具，就可以在多个项目中使用。

Head First: 就没有什么缺点吗？

O/S: 老实说，小工具长得不好看，它们通常在命令行下工作，因此没有吸引眼球的界面。

Head First: 这个影响大吗？

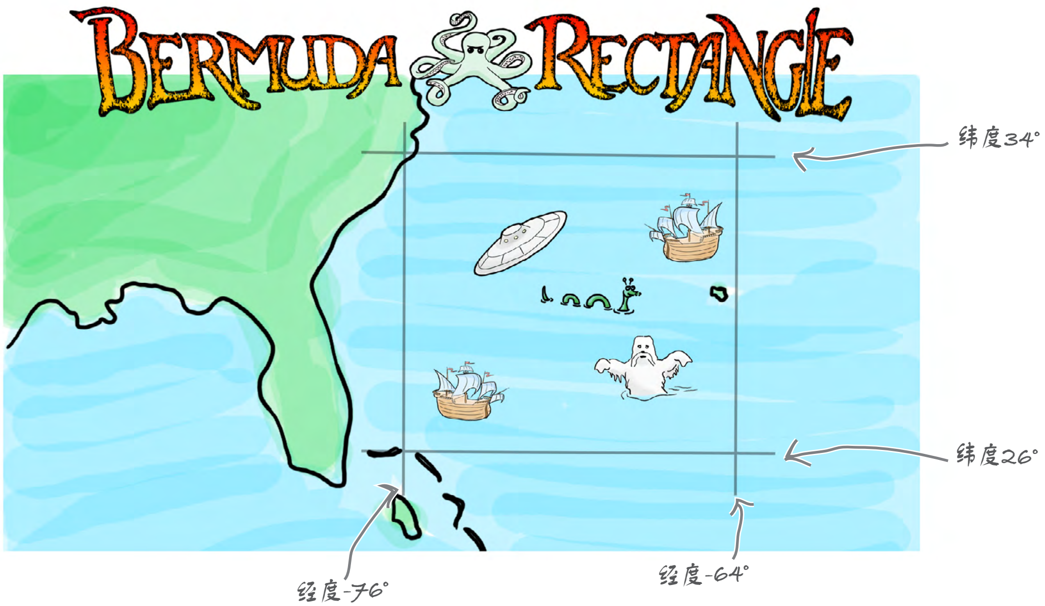
O/S: 不大，无论是桌面应用程序还是网站，只要你用小工具实现了程序的核心部分，就可以把它们连接到一个好看的界面上。嘿，时间到了，不好意思了，我要抢占你，让其他进程上台。

Head First: 好的，谢谢你，O/S。非常高兴……呼……呼……呼……

灵活的小工具

小工具的优点之一是灵活。如果有一个程序，它很好地完成了一件事，那么就可以在很多场合用到它。打个比方，假如你创建了一个在文件中搜索文本的程序，就可以在很多地方用到它。

例如geo2json工具，你创建它是为了显示骑行路线，但为什么不能用它来做其他事情？比如导航……



为了见识我们的工具有多灵活，我们用它解决一个完全不同的问题。刚才我们的程序从GPS读取数据，然后全部显示在地图上，这次我们提高难度，只显示落在百慕大三角内的数据。

也就是说只显示符合以下条件的数据：

```
((latitude > 26) && (latitude < 34))

((longitude > -76) && (longitude < -64))
```

你准备从哪里下手？

切莫修改geo2json工具

我们的geo2json工具会显示所有数据，应不应该修改geo2json，好让它在输出之前先检查数据？

我们当然可以这么做，但别忘了，小工具：

做一件事并把它做好

你不希望修改geo2json工具，因为你想让它只做一件事。如果让程序做了更复杂的事，会给老用户带来麻烦。

如果不想修改geo2json工具，该怎么做？

小工具设计Tips

设计像geo2json这样的小工具时，应遵循以下原则：

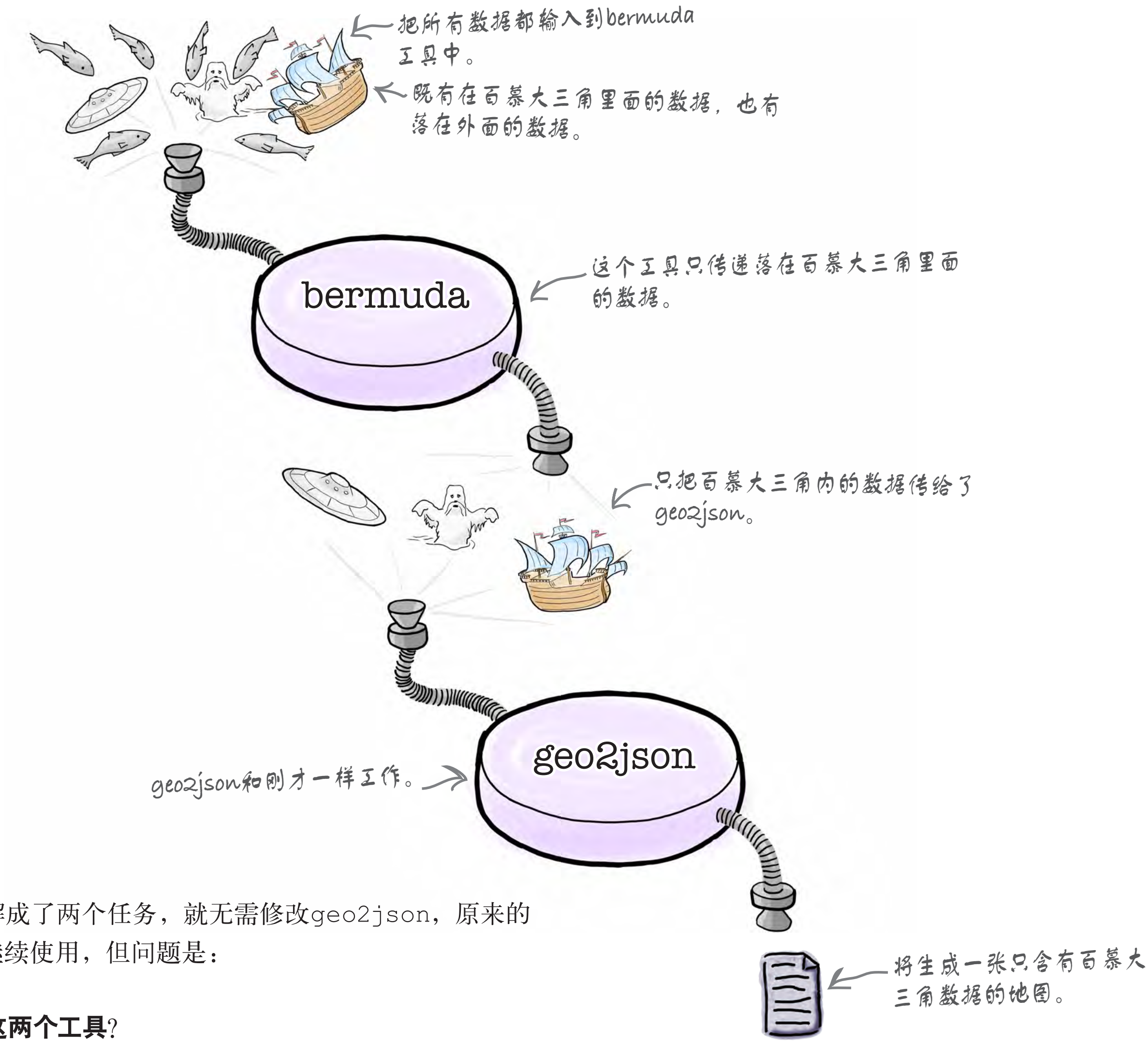
* 从标准输入读取数据。
* 在标准输出显示数据。
* 处理文本数据，而不是难以阅读的二进制格式。
* 只做一件简单的事。

一个任务对应一个工具

如果想要跳过百慕大三角以外的数据，应该再创建一个工具来做这件事。

你将有**两个**工具，一个是新的**bermuda**工具，它过滤百慕大三角以外的数据；另一个是原来的geo2json工具，它将剩余数据转化成地图所需要的格式。

像这样连接两个程序：



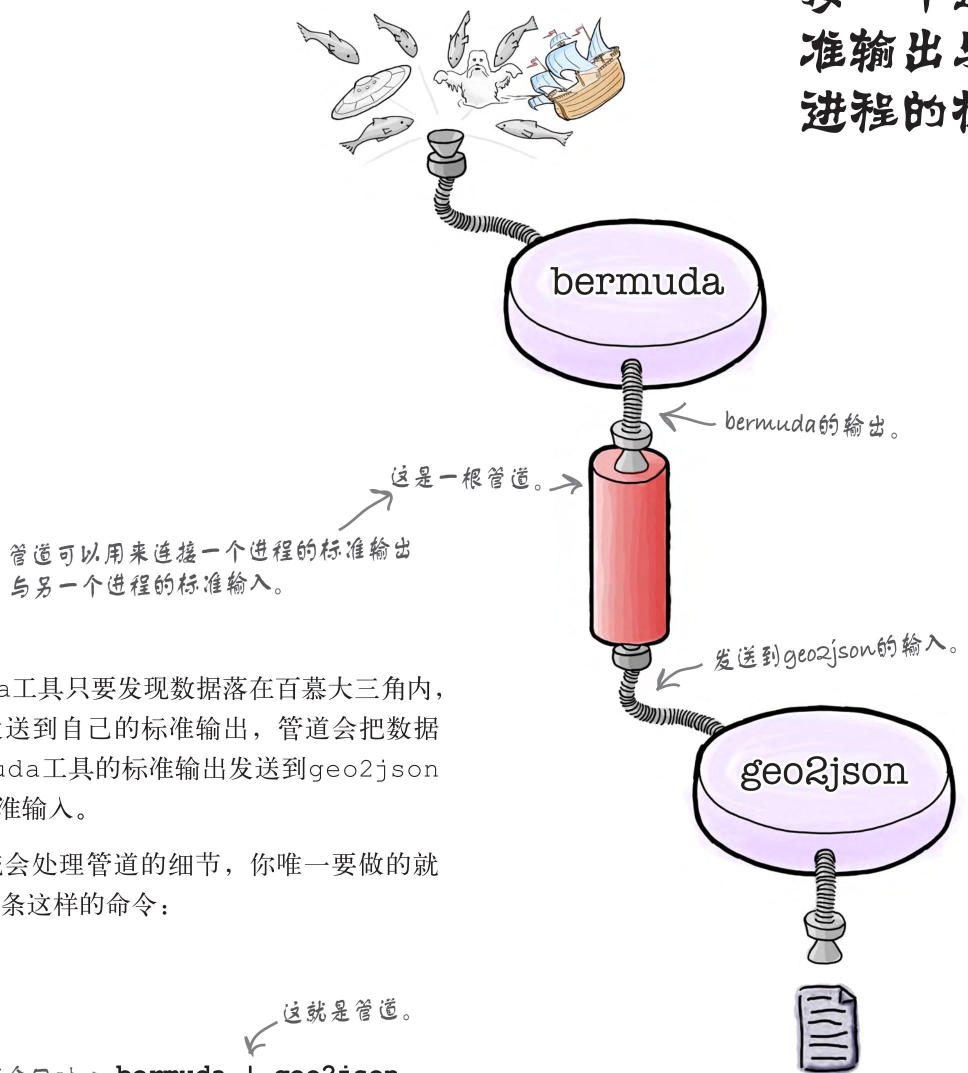
把问题分解成了两个任务，就无需修改geo2json，原来的用户还能继续使用，但问题是：

如何连接这两个工具？

用管道连接输入与输出

你已经知道如何使用重定向连接同一个程序的标准输入和标准输出，但现在要把**bermuda**工具的标准输出连接到**geo2json**工具的标准输入，像这样：

符号 | 表示管道(pipe)，它能连接一个进程的标准输出与另一个进程的标准输入。



bermuda工具只要发现数据落在百慕大三角内，就把它发送到自己的标准输出，管道会把数据从bermuda工具的标准输出发送到geo2json工具的标准输入。

操作系统会处理管道的细节，你唯一要做的就是输入一条这样的命令：

```
bermuda | geo2json
```

这就是管道。

操作系统会同时运行两个程序。

bermuda的输出会变成geo2json的输入。

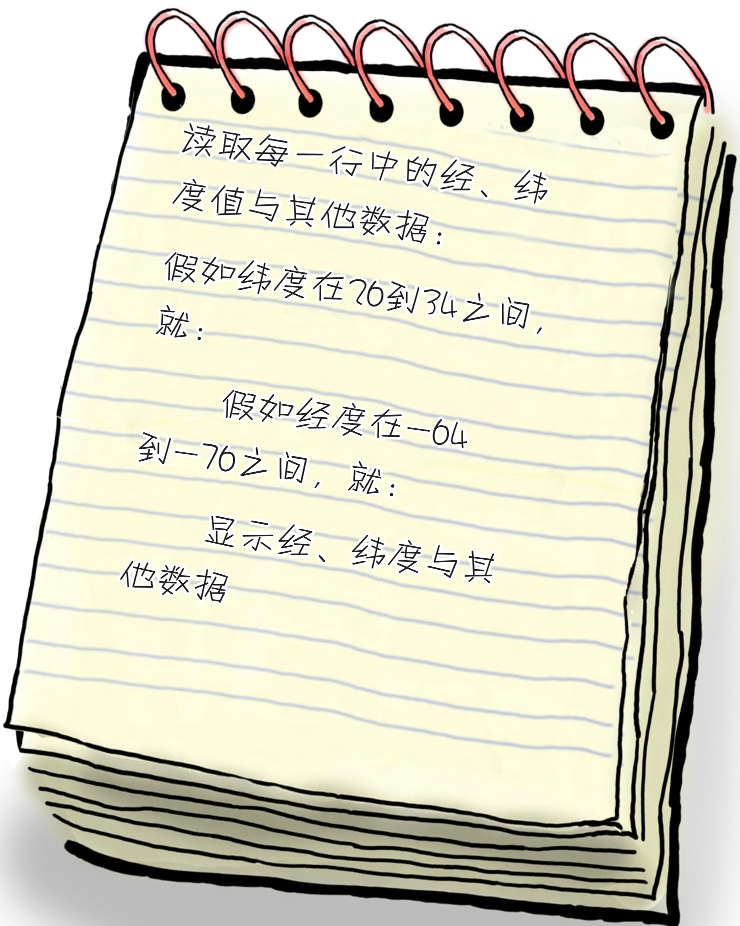
现在是时候建立bermuda工具了。

bermuda工具

bermuda和geo2json的工作方式极其相似，它们都逐行读取GPS数据，然后向标准输出发送数据。

但有两点不同：首先，bermuda工具不会把所有数据都发送到标准输出中，而只发送那些落在百慕大三角内的数据；其次，bermuda工具输出、输入数据的格式相同，都是用来保存GPS数据的CSV格式。

下面就是bermuda工具的伪代码：



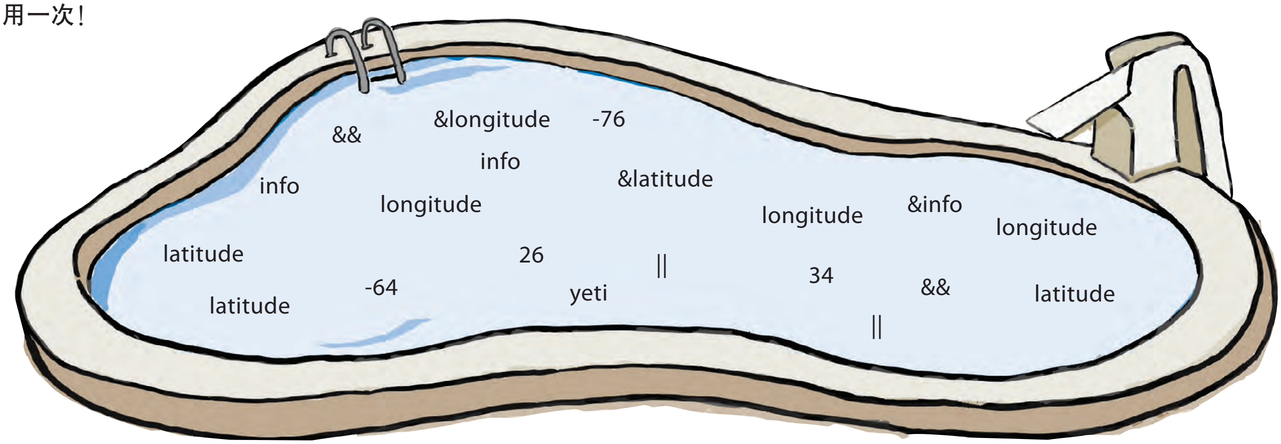
我们来把伪代码变成C语言。

游泳池拼图

你需要补全bermuda程序的代码，取出游泳池中的碎片，把它们填到空白的横线处，有的碎片可能一次都用不到。

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  while (scanf("%f,%f,%79[^\n]",................,................,................) ==  3)
    if ((................ > ................) ................ (................ < ................))
      if ((................ > ................) ................ (................ < ................))
        printf("%f,%f,%s\n",................,................,................);

  return 0;
}
```

注意：每样东西只能使用一次！

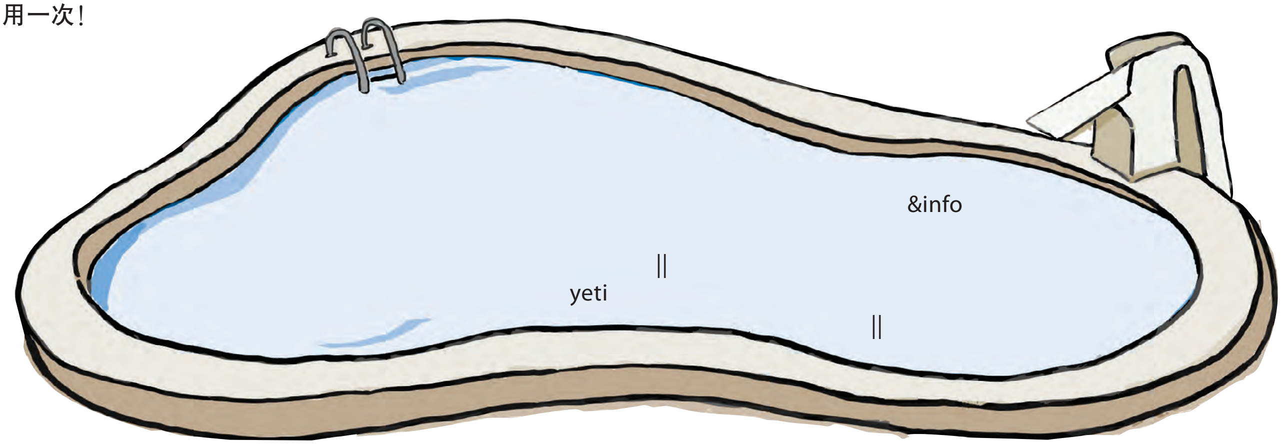


游泳池拼图解答

你取出游泳池中的碎片，补全了bermuda程序的代码，把它们填到了空白的横线处。

```
#include <stdio.h>

int main()
{
  float latitude;
  float longitude;
  char info[80];
  while (scanf("%f,%f,%79[^\n]", &latitude, &longitude, info) == 3)
    if ((latitude > 26) && (latitude < 34))
      if ((longitude > -76) && (longitude < -64))
        printf("%f,%f,%s\n", latitude, longitude, info);

  return 0;
}
```

注意：每样东西只能使用一次！



试驾

动起来！

完成了bermuda工具，下面就和geo2json工具一起使用，看看能不能把百慕大三角以外的数据都过滤掉。

编译这两个工具，打开控制台，输入以下命令同时运行两个程序：

spooky.csv文件的下载地址：
http://chengyichao.info/hfc/spooky.csv

```
(./bermuda | ./geo2json) < spooky.csv > output.json
```

别忘了，如果在Windows中，别加“./”。

这是连接两个进程的管道。

所有数据都在这个文件中。

当把两个程序连在一起，就可以把它们看成一个程序。

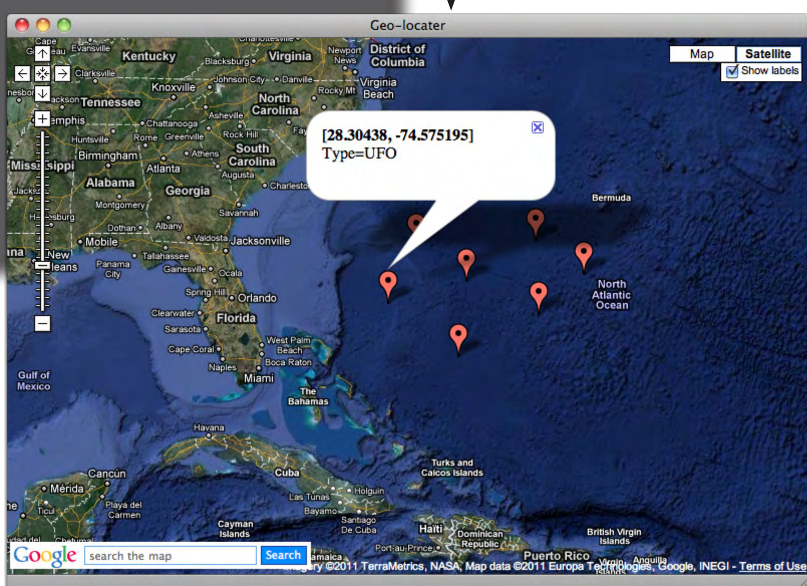
bermuda工具过滤我们不想要的数据。

geo2json工具把数据转化为JSON格式。

我们把输出保存在这个文件中。

两个独立的程序用管道连接以后就可以看成一个程序，可以重定向它的标准输入和标准输出。

```
File Edit Window Help MyAngle
> (./bermuda | ./geo2json) < spooky.csv > output.json
```



好极了，程序正确运行了！

这里没有蠢问题

问：为什么小工具要使用标准输入和标准输出？

答：有了它们，就可以轻易用管道将小工具们串连起来。

问：为什么要把它们串连在一起？

答：小工具只能解决一个小技术问题，例如转换数据的格式，而无法解决整个问题。只有把它们组合在一起，才能解决大问题。

问：到底什么是管道？

答：不同操作系统实现管道的方法不同，可能用存储器，也可能用临时文件。我们只要知道它从一端接收数据，在另一端发送数据就行了。

问：如果两个程序用管道相连，第二个程序要不要等第一个程序执行完后才能开始运行？

答：不需要，两个程序可以同时运行，第一个程序一发出数据，第二个程序马上就可以处理。

问：为什么小工具要使用文本？

答：文本是一种开放格式，程序员可以用文本编辑器来查看小工具的输出，并理解里面的内容，相比之下，二进制格式就难懂多了。

问：我能用管道连接多个程序吗？

答：能啊，只要在每个程序前加上一个|就行了，一连串相连的进程就叫流水线（pipeline）。

问：当我用管道连接多个进程时，< 与 > 分别重定向哪个进程的标准输入、哪个进程的标准输出？

答：< 会把文件内容发送到流水线中第一个进程的标准输入，> 会捕获流水线中最后一个进程的标准输出。

问：当我在命令行中运行`bermuda`和`geo2json`程序时，它们外面的括号是必需的吗？

答：是的，这对括号保证了数据文件由`bermuda`程序的标准输入来读取。

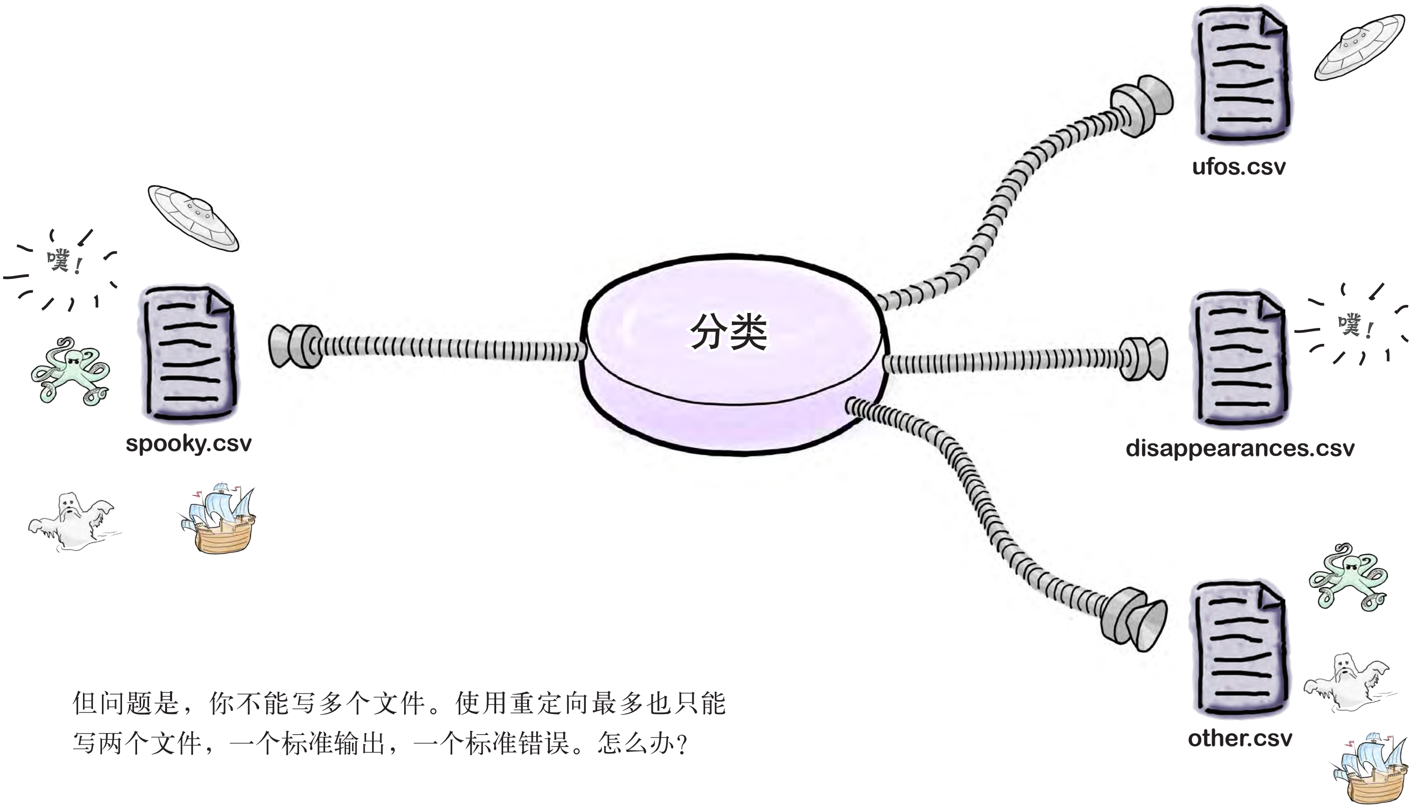
要点

- 如果想完成一个不同的任务，应该另外写一个小工具。
- 小工具应该使用标准输入和标准输出。
- 小工具通常读写文本数据。
- 可以用管道连接一个进程的标准输出和另一个进程的标准输入。

输出多个文件

我们已经会用重定向从文件中读取数据，然后写到另一个文件中。但如果程序需要做更复杂的事情怎么办？例如向**多个文件**发送数据。

假设你需要创建另一个工具，它从文件中读取一批数据，然后将数据分类，写到多个文件。



但问题是，你不能写多个文件。使用重定向最多也只能写两个文件，一个标准输出，一个标准错误。怎么办？

创建自己的数据流

程序运行时，操作系统会为它创建三条数据流：标准输入、标准输出和标准错误。但有时你需要创建自己的数据流。

好在操作系统没有规定只能使用它分配的三条数据流，你可以在程序运行时创建自己的数据流。

每条数据流用一个指向文件的指针来表示，可以用**fopen()**函数创建新数据流。

将创建一条数据流，从文件中读取数据。这是文件名。这是模式，r表示“读”（read）。

```
FILE *in_file = fopen("input.txt", "r");
```

将创建一条数据流，向文件写数据。这是文件名。这是模式：w表示“写”（write）。

```
FILE *out_file = fopen("output.txt", "w");
```

fopen()函数接收两个参数：文件名和模式。共有三种模式，分别是**w**（写文件）、**r**（读文件）与**a**（在文件末尾追加数据）。

三种模式分别是：
“w” = 写（write）
“r” = 读（read）
“a” = 追加（append）

创建数据流后，可以用**fprintf()**往数据流中打印数据。如果想要从文件中读取数据，则可以用**fscanf()**函数：

```
fprintf(out_file, "不要穿 %s 色的衣服和 %s 色的裤子", "红", "绿");

fscanf(in_file, "%79[^\n]\n", sentence);
```

最后，当用完数据流，别忘了关闭它。虽然所有的数据流在程序结束后都会自动关闭，但你仍应该自己关闭它们：

```
fclose(in_file);
fclose(out_file);
```

现在试一试。

磨笔上阵

下面这段程序代码将从GPS文件读取所有数据，写到其他三个文件中的一个，看看你能不能将空格填满。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
  char line[80];
  FILE *in = fopen("spooky.csv", ...............);
  FILE *file1 = fopen("ufos.csv", ...............);
  FILE *file2 = fopen("disappearances.csv", ...............);
  FILE *file3 = fopen("other.csv", ...............);
  while (...............(in, "%79[^\n]\n", line) == 1) {
    if (strstr(line, "UFO"))
      ...............(file1, "%s\n", line);
    else if (strstr(line, "Disappearance"))
      ...............(file2, "%s\n", line);
    else
      ...............(file3, "%s\n", line);
  }
  ...............(file1);
  ...............(file2);
  ...............(file3);
  return 0;
}
```

这里没有蠢问题

问： 最多能有几条数据流？

答： 这取决于操作系统。通常情况下，一个进程最多可以有256条数据流。但请记住，数据流的数量是有限的，用完后应该关闭它们。

问： 为什么FILE要大写？

答： 说来话长，最早FILE是用宏定义的，而宏的名字通常都要大写。稍后会看到宏。

磨笔上阵 解答

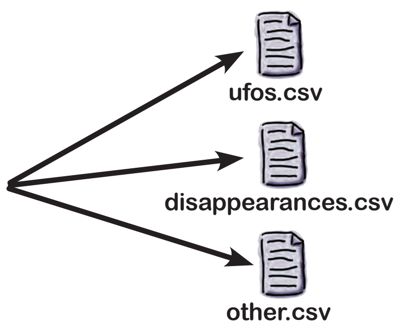
下面这段程序代码将从GPS文件读取所有数据，写到其他三个文件中的一个，你将填满空格。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main()
{
  char line[80];
  FILE *in = fopen("spooky.csv", "r");
  FILE *file1 = fopen("ufos.csv", "w");
  FILE *file2 = fopen("disappearances.csv", "w");
  FILE *file3 = fopen("other.csv", "w");
  while (fscanf(in, "%79[^\n]\n", line) == 1) {
    if (strstr(line, "UFO"))
      fprintf(file1, "%s\n", line);
    else if (strstr(line, "Disappearance"))
      fprintf(file2, "%s\n", line);
    else
      fprintf(file3, "%s\n", line);
  }
  fclose(file1);
  fclose(file2);
  fclose(file3);
  return 0;
}
```

程序运行了，但是……

当你用以下命令编译并运行程序：

gcc categorize.c -o categorize && ./categorize



程序会逐行读取*spooky.csv*文件中的数据，分别写到*ufos.csv*、*disappearances.csv*、*other.csv*这三个文件中。

程序虽然正确运行了，但如果用户想改变分类方法怎么办？例如用户想搜索别的关键字，或把数据写到其他文件中。有没有什么办法能让用户设置关键字与文件，又不必重新编译程序？

main()可以做得更多

用户有权修改程序的工作方式。对GUI程序来说，可以修改程序的首选项；而对于categorize这样的命令行程序，可以传给它命令行参数。

第一个要过滤的词。

所有mermaid数据都将保存在这个文件中。

表示你想搜索Elvis。

```
./categorize mermaid mermaid.csv Elvis elvises.csv the_rest.csv
```

其余数据保存在这个文件中。

所有与Elvis有关的数据将保存在这里。

怎么才能在程序中读取命令行参数呢？之前创建的所有**main()**函数都没有参数。main()函数其实有两种形式，第二种如下：

```
int main(int argc, char *argv[])
{
   .... 做事情....
}
```

main()函数能以字符串数组的形式读取命令行参数。由于C语言没有内置字符串，所谓的字符串数组其实是一个字符指针数组，像这样：

"./categorize"	"mermaid"	"mermaid.csv"	"Elvis"	"elvises.csv"	"the_rest.csv"
argv[0]。	argv[1]。	argv[2]。	argv[3]。	argv[4]。	argv[5]。

第一个参数是要运行的程序的名字。

在C语言中，需要想办法知道数组的长度，所以main()函数有两个参数，argc的值用来记录数组中元素的个数。

命令行参数可以让程序更灵活，如果用户能调整程序的工作方式，就会觉得程序很有用。

下面就来修改categorize程序，让它变得更灵活。

用户运行程序时，命令行中第一个参数是程序名。

也就是说，第一个命令行参数其实是argv[1]。

代码冰箱贴

这是修改以后的categorize工具，程序从命令行读取搜索关键字和使用的文件，你能否把冰箱贴放到正确的位置？

用以下命令运行程序：

```
./categorize mermaid mermaid.csv Elvis elvises.csv the_rest.csv
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char line[80];

  if (.................. != ..................) {
    fprintf(stderr, "You need to give 5 arguments\n");
    return 1;
  }
  FILE *in = fopen("spooky.csv", "r");

  FILE *file1 = fopen(.................., "w");

  FILE *file2 = fopen(.................., "w");

  FILE *file3 = fopen(.................., "w");
```

```
  while (fscanf(in, "%79[^\n]\n", line) == 1) {

    if (strstr(line, ..................))
      fprintf(file1, "%s\n", line);

    else if (strstr(line, ..................))
      fprintf(file2, "%s\n", line);
    else
      fprintf(file3, "%s\n", line);
  }
  fclose(file1);
  fclose(file2);
  fclose(file3);
  return 0;
}
```

6

5

argv[5]

argc

argv[2]

argv[4]

argv[1]

argv[3]

代码冰箱贴解答

这是修改以后的categorize工具，程序从命令行读取搜索关键字和使用的文件，请把冰箱贴放到正确的位置。

用以下命令运行程序：

```
./categorize mermaid mermaid.csv Elvis elvises.csv the_rest.csv
```

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

int main(int argc, char *argv[])
{
  char line[80];

  if ( argc != 6 ) {
    fprintf(stderr, "You need to give 5 arguments\n");
    return 1;
  }
  FILE *in = fopen("spooky.csv", "r");

  FILE *file1 = fopen( argv[2] , "w");

  FILE *file2 = fopen( argv[4] , "w");

  FILE *file3 = fopen( argv[5] , "w");
```

```
  while (fscanf(in, "%79[^\n]\n", line) == 1) {

    if (strstr(line, argv[1]))
      fprintf(file1, "%s\n", line);

    else if (strstr(line, argv[3]))
      fprintf(file2, "%s\n", line);
    else
      fprintf(file3, "%s\n", line);
  }
  fclose(file1);
  fclose(file2);
  fclose(file3);
  return 0;
}
```

5

试驾

好了，我们来试一下新的代码，你需要一个叫*spooky.csv*的测试文件。

```
30.685163,-68.137207,Type=Yeti
28.304380,-74.575195,Type=UFO
29.132971,-71.136475,Type=Ship
28.343065,-62.753906,Type=Elvis
27.868217,-68.005371,Type=Goatsucker
30.496017,-73.333740,Type=Disappearance
26.224447,-71.477051,Type=UFO
29.401320,-66.027832,Type=Ship
37.879536,-69.477539,Type=Elvis
22.705256,-68.192139,Type=Elvis
27.166695,-87.484131,Type=Elvis
```

spooky.csv

运行`categorize`时，要用几个命令行参数告诉它查找哪些关键字以及使用哪些文件名：

File Edit Window Help ThankYouVeryMuch

```
> categorize UFO aliens.csv Elvis elvises.csv the_rest.csv
```

程序运行以后，生成了以下文件：

```
28.304380,-74.575195,Type=UFO
26.224447,-71.477051,Type=UFO
```

aliens.csv

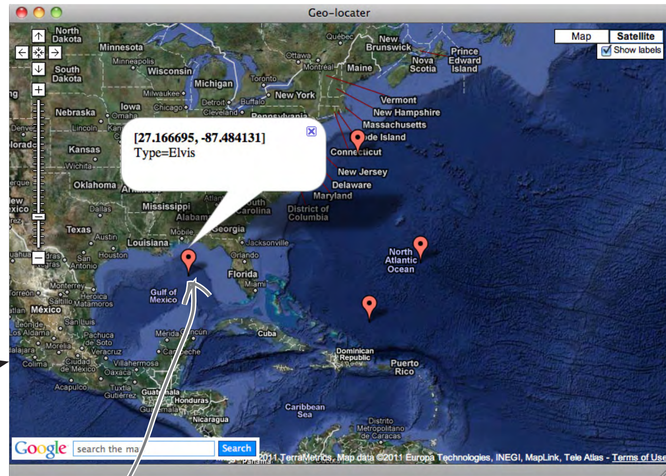
```
30.685163,-68.137207,Type=Yeti
29.132971,-71.136475,Type=Ship
27.868217,-68.005371,Type=Goatsucker
30.496017,-73.333740,Type=Disappearance
29.401320,-66.027832,Type=Ship
```

the_rest.csv

```
28.343065,-62.753906,Type=Elvis
37.879536,-69.477539,Type=Elvis
22.705256,-68.192139,Type=Elvis
27.166695,-87.484131,Type=Elvis
```

elvises.csv

用geo2json工具转化elvises.csv的格式，然后用地图显示。



Elvis离开了大楼。

安全检查

在Head First实验室，我们从来不会犯错（咳咳）。但是在现实世界，当你在程序中打开文件准备读写时，最好检查一下有没有错误发生。好在如果数据流打开失败，fopen()函数会返回0，也就是说如果想检查错误，可以将下面这段代码：

```
FILE *in = fopen("我不存在.txt", "r");
```

改成这样：

```
FILE *in;
if (!(in = fopen("我不存在.txt", "r"))) {
  fprintf(stderr, "无法打开文件.\n");
  return 1;
}
```

Head First披萨屋耳闻

十个程序有九个需要选项。聊天程序有“系统设置”，游戏有调整难度的选项，而命令行工具需要有命令行选项。

命令行选项是一些小开关，它们经常出现在命令行工具中：

```
ps -ae
```

显示所有进程，包括后台运行的进程。

```
tail -f logfile.out
```

持续显示文件末尾新添加的数据。

由库代劳

很多程序都会使用命令行选项，因此有一个专门的库函数，可以用它来简化处理过程。这个库函数叫**getopt()**，每一次调用都会返回命令行中下一个参数。

看一下它是怎么工作的，假设程序能够接收一组不同的选项：

使用4台引擎。　开启“无敌模式”。

```
rocket_to -e 4 -a Brasilia Tokyo London
```

程序需要两个选项，一个选项接收值，-e代表“引擎”；另一个选项代表了*开或关*，-a代表“无敌模式”。可以循环调用getopt()来处理这两个选项，像这样：

```
#include <unistd.h>

...

while ((ch = getopt(argc, argv, "ae:")) != EOF)
  switch(ch) {
  ...
  case 'e':
    engine_count = optarg;
  ...
  }

argc -= optind;
argv += optind;
```

需要包含这个头文件。

表示选项a和e是有效的。

这是处理每个选项的代码。

“:”表示选项e需要一个参数。

在这里读取e选项的参数。

optind保存了“getopt()函数从命令行读取了几个选项”。

最后这两行用来跳过已读取的选项。

C标准礼貌指南

*unistd.h*头文件不属于C标准库，而是POSIX库中的一员。POSIX的目标是创建一套能够在所有主流操作系统上使用的函数。

在循环中，用switch语句处理每个有效选项。字符串**ae:**告诉getopt()函数“a和e是有效选项”，e后面的冒号表示“-e后面需要再跟一个参数”，getopt()会用**optarg**变量指向这个参数。

循环结束以后，为了让程序读取命令行参数，需要调整一下argv和argc变量，跳过所有选项，最后argv数组将变成这样：

```
Brasilia Tokyo London
```

argv[0]。　argv[1]。　argv[2]。

经过一番处理，0号参数不再是程序名了。

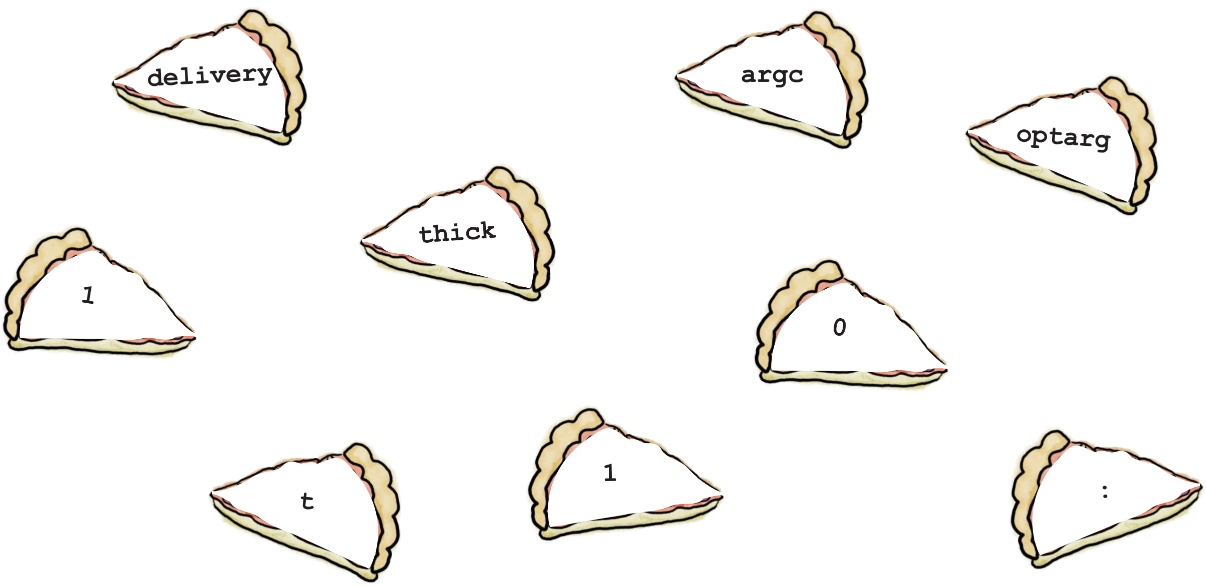
argv[0]会指向选项后的第一个命令行参数。

披萨拼图

有人偷吃了我们的代码披萨，你能补全披萨并复原order_pizza程序吗？

```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  char *delivery = "";
  int thick = 0;
  int count = 0;
  char ch;


  while ((ch = getopt(argc, argv, "d ............................ ............................")) != EOF)
    switch (ch) {
    case 'd':


      ............................ = ............................ ;
      break;
    case 't':


      ............................ = ............................ ;
      break;
    default:
      fprintf(stderr, "Unknown option: '%s'\n", optarg);


      return ............................ ;
    }
```

```
  argc -= optind;
  argv += optind;

  if (thick)
    puts("Thick crust.");

  if (delivery[0])
    printf("To be delivered %s.\n", delivery);

  puts("Ingredients:");



  for (count = ............................; count < ............................; count++)
    puts(argv[count]);
  return 0;
}
```



披萨拼图解答

有人偷吃了我们的代码披萨，请补全披萨并复原order_pizza程序。

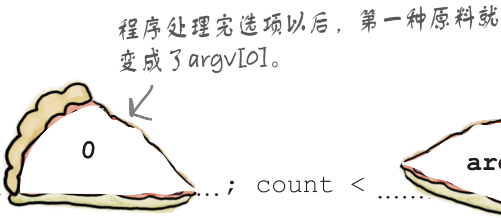
```
#include <stdio.h>
#include <unistd.h>

int main(int argc, char *argv[])
{
  char *delivery = "";
  int thick = 0;
  int count = 0;
  char ch;

  while ((ch = getopt(argc, argv, "d:t")) != EOF)
    switch (ch) {
    case 'd':
      delivery = optarg;
      break;
    case 't':
      thick = 1;
      break;
    default:
      fprintf(stderr, "Unknown option: '%s'\n", optarg);
      return 1;
    }
```

d后面跟一个冒号，因为d选项要接收参数。

我们把delivery变量指向d选项提供的参数。

别忘了，在C语言中把变量设为1等价于令它为真。

```
  argc -= optind;
  argv += optind;

  if (thick)
    puts("Thick crust.");

  if (delivery[0])
    printf("To be delivered %s.\n", delivery);

  puts("Ingredients:");

  for (count = 0; count < argc; count++)
    puts(argv[count]);
  return 0;
}
```



试驾

现在就试试“点披萨”程序吧：

编译程序。

前两次你一个选项都没用。

然后试试d选项，给它一个参数now。

再试试t选项，别忘了，t选项不接收任何参数。

最后试试跳过d选项后面的参数，结果产生了错误。

```
File  Edit  Window  Help  Anchovies?
> gcc order_pizza.c -o order_pizza
> ./order_pizza Anchovies
Ingredients:
Anchovies
> ./order_pizza Anchovies Pineapple
Ingredients:
Anchovies
Pineapple
> ./order_pizza -d now Anchovies Pineapple
To be delivered now.
Ingredients:
Anchovies
Pineapple
> ./order_pizza -d now -t Anchovies Pineapple
Thick crust.
To be delivered now.
Ingredients:
Anchovies
Pineapple
> ./order_pizza -d
order_pizza: option requires an argument -- d
Unknown option: '(null)'
>
```

程序正确运行了！

好啦，这一章你学了不少东西，不但深入理解了标准输入、标准输出和标准错误，而且学会了使用重定向和自己创建的数据流读写文件。最后，还学会了处理命令行参数和选项。

C程序员不是在创造小工具，就是在创造小工具的路上，Linux中大部分小工具都是用C语言写的。精心设计小工具，确保它们做一件事并把它做好。假以时日，你就能成为一名卓越的C程序员。

这里没有蠢问题

问： 我能合并两个选项吗？例如用`-td now`代替`-d now -t`。

答： 可以，getopt()函数会全权处理它们。

问： 我可以改变选项之间的顺序吗？

答： 可以，因为我们用循环读取选项，所以-d now -t、-t -d now、-td now都一样。

问： 也就是说，只要程序在命令行看到一个前缀为－值，就会把它当成选项处理？

答： 是的，前提是它必须在命令行参数之前出现。

问： 如果我想在命令行参数使用负数怎么办？像`set_temperature -c -4`，程序会把4当作选项吗？

答： 为了避免歧义，可以用--隔开参数和选项，比如set_temperature -c -- -4。getopt()看到--就会停止读取选项，程序会把后面的内容当成普通的命令行参数读取。

要点

- main()函数有两个版本，一个有命令行参数，一个没有。
- 命令行参数通过两个变量传递给main()函数，一个是参数的计数，另一个是指针（指向参数字符串）数组。
- 命令行选项是以“-”开头的命令行参数。
- getopt()函数会帮助你处理命令行选项。
- 为了定义有效的选项，可以传给getopt()一个字符串，例如ae:。
- 选项之后的“:”（冒号）表示该选项需要接收一个参数。
- getopt()会用optarg变量记录选项参数。
- 读取完全部的选项以后，应该用optind变量跳过它们。

C语言工具箱

学完第3章，现在你的工具箱中又多出了小工具。关于本书提示工具条的完整列表，请见附录ii。

printf()和scanf()使用标准输出和标准输入来交互。

标准输出默认在显示器上显示数据。

标准错误专门用来输出错误消息。

标准输入默认从键盘读取数据。

可以用重定向把标准输入、标准输出和标准错误连接到其他地方。

可以用fprintf(stderr,...)把数据打印到标准错误。

命令行参数以字符串指针数组的形式传递给main()。

可以用fopen("文件名",模式)创建你自己的数据流。

三种模式分别是w（写入）、r（读取）、a（追加）。

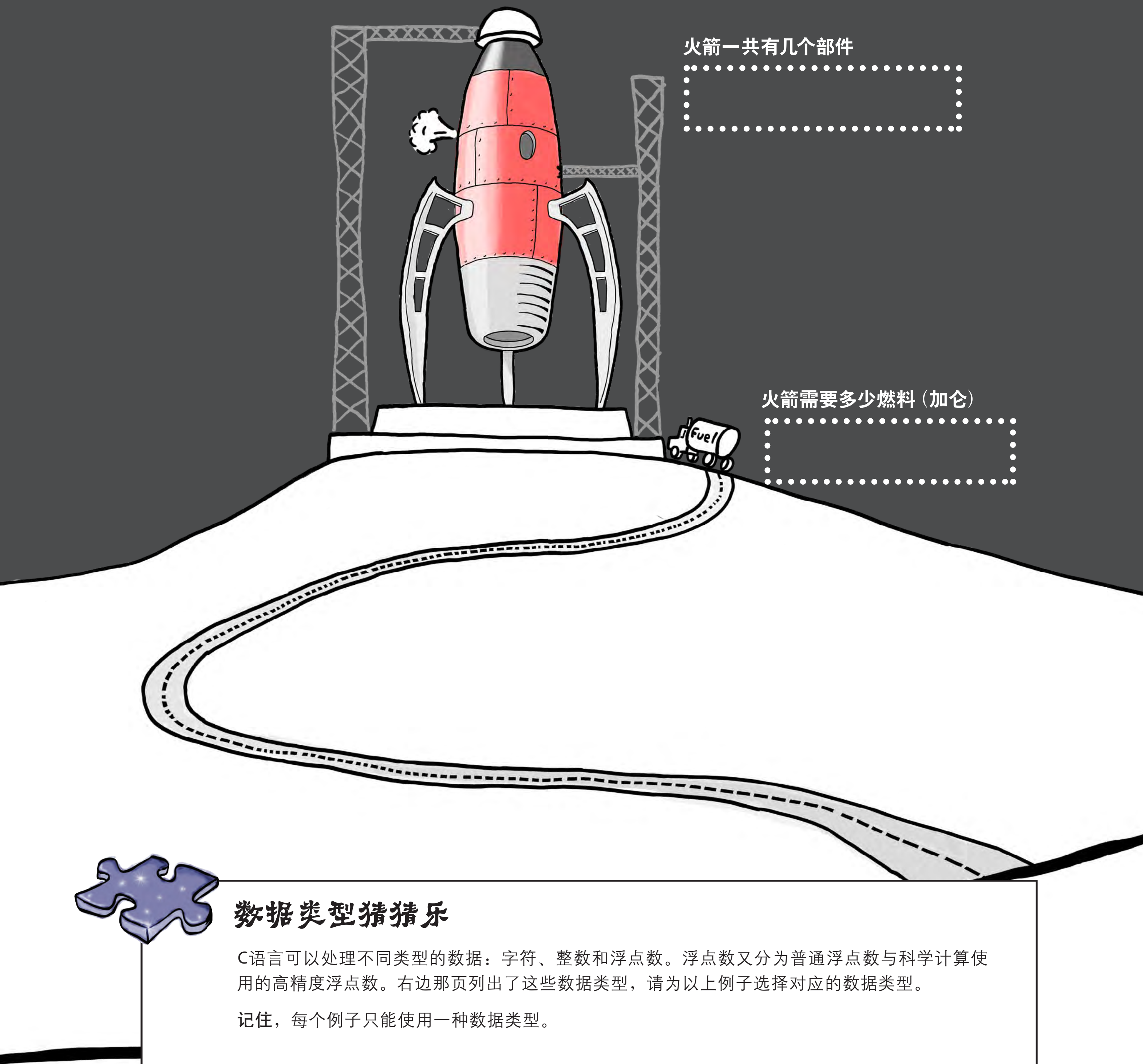
用getopt()函数读取命令行选项很方便。

4 使用多个源文件

分而治之

大程序不等于大源文件。

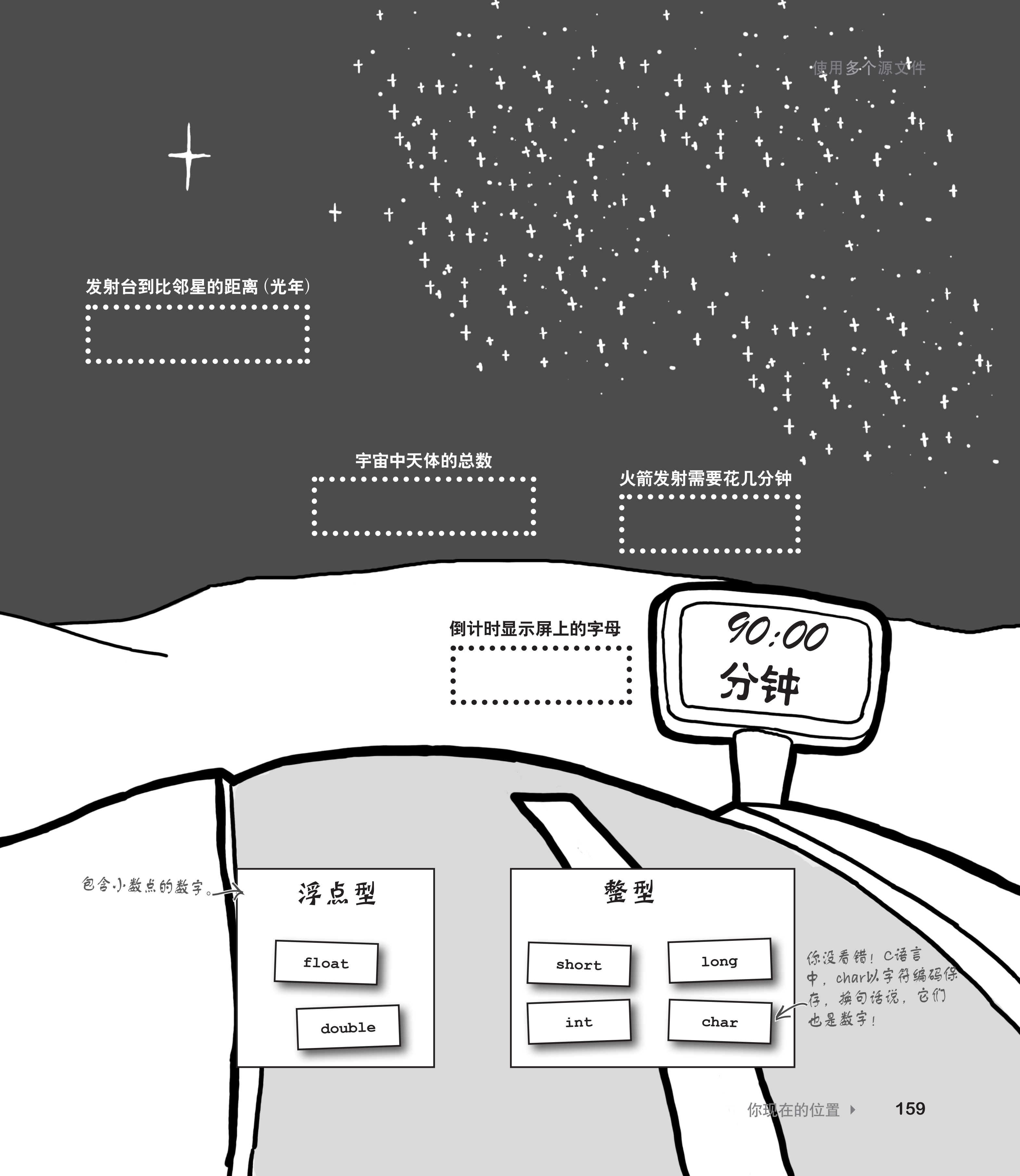
你能想象一个企业级的程序如果只有一个源文件，维护起来有多么困难与耗时吗？在本章中，你将学习怎样把源代码分解为易于管理的小模块，然后把它们合成一个大程序，同时还将了解数据类型的更多细节，并结识一个新朋友：**make**。

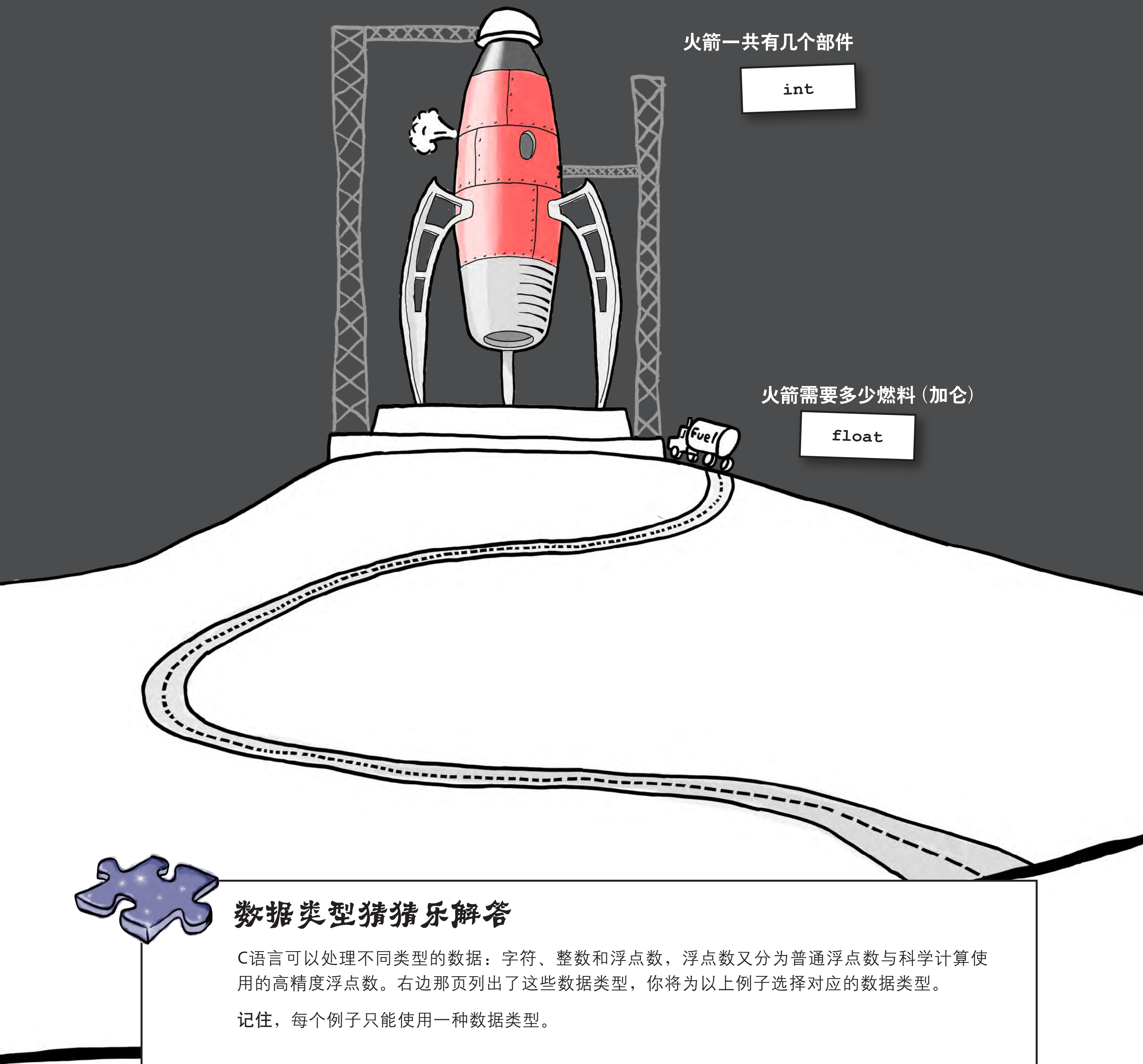


数据类型猜猜乐

C语言可以处理不同类型的数据：字符、整数和浮点数。浮点数又分为普通浮点数与科学计算使用的高精度浮点数。右边那页列出了这些数据类型，请为以上例子选择对应的数据类型。

记住，每个例子只能使用一种数据类型。

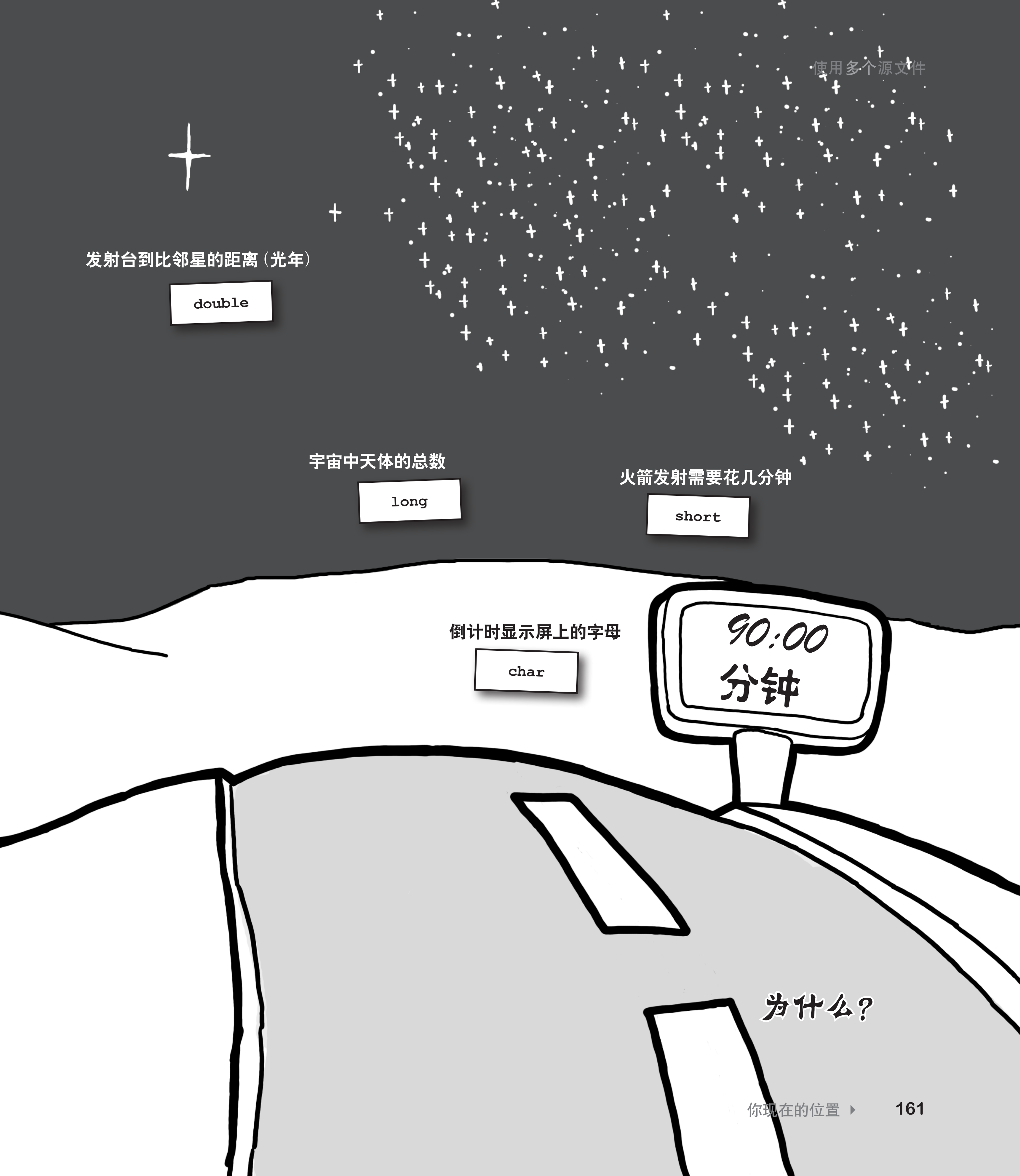

发射台到比邻星的距离（光年）
宇宙中天体的总数
火箭发射需要花几分钟
倒计时显示屏上的字母
90:00
分钟
包含小数点的数字。
浮点型
float
double
整型
short
long
int
char
你没看错！C语言中，char以字符编码保存，换句话说，它们也是数字！



数据类型猜猜乐解答

C语言可以处理不同类型的数据：字符、整数和浮点数，浮点数又分为普通浮点数与科学计算使用的高精度浮点数。右边那页列出了这些数据类型，你将为以上例子选择对应的数据类型。

记住，每个例子只能使用一种数据类型。


发射台到比邻星的距离（光年）
double
宇宙中天体的总数
long
火箭发射需要花几分钟
short
倒计时显示屏上的字母
char
90:00
分钟
为什么？

简明数据类型指南

char

字符在计算机的存储器中以字符编码的形式保存，字符编码是一个数字，因此在计算机看来，A与数字65完全一样。

65是A的ASCII码。

int

如果你要保存一个整数，通常可以使用int。不同计算机中int的大小不同，但至少应该有16位。一般而言，int可以保存几万以内的数字。

short

但有时你想节省一点空间，毕竟如果只想保存一个几百、几千的数字，何必用int？可以用short，short通常只有int的一半大小。

long

但如果想保存一个很大的计数呢？long数据类型就是为此而生的。在某些计算机中，long的大小是int的两倍，所以可以保存几十亿以内的数字；但大部分计算机的long和int一样大，因为在这些计算机中int本身就很大。long至少应该有32位。

float

float是保存浮点数的基本数据类型。平时你会碰到很多浮点数，比如一杯香橙摩卡冰乐有多少毫升，就可以用float保存。

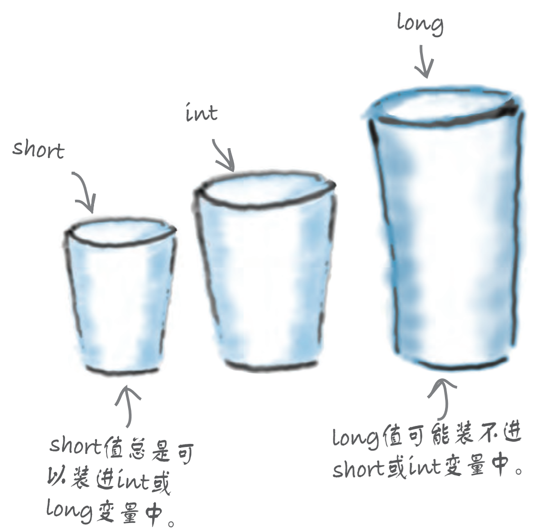
double

但如果想表示很精确的浮点数呢？如果想让计算结果精确到小数点以后很多位，可以使用double。double比float多占一倍空间，可以保存更大、更精确的数字。

勿以小杯盛大物

赋值时要保证值的类型与保存它的变量类型相匹配。

不同数据类型的大小不同，千万别让值的大小超过变量。short比int的空间小，int又比long小。



完全可以在int或long变量中保存short值。因为变量有足够的空间，你的代码将正确运行：

```
short x = 15;
int y = x;
printf("y的值是%i\n", y);
```

这表明y = 15。

但是反过来，比如你想在short变量中保存int值，就不行。

```
int x = 100000;
short y = x;
print("y的值是%hi\n", y);
```

%hi用来格式化short值。

有时，编译器能发现你想在小变量中保存大值，然后给出一条警告，但大多数情况下编译器不会发现。这时当你运行代码，计算机无法在short变量中保存100 000。计算机能装多少0、1就装多少，而最终保存在变量y中的数字已面目全非：

```
y的值是 = -31072
```

百宝箱

为什么把一个很大的数保存到short中会变成负数？数字以二进制保存，二进制的100 000看起来像这样：

```
x <- 0001 1000 0110 1010 0000
```

当计算机想把这个值保存到short时，发现只能保存2个字节，所以只保存了数字右半边：

```
y <- 1000 0110 1010 0000
```

最高位是1的二进制有符号数会被当成负数处理，它等价于下面的十进制数：

```
-31072
```

使用类型转换把float值存进整型变量

你认为下面这段代码将显示什么?

```
int x = 7;
int y = 2;
float z = x / y;
printf("z = %f\n", z);
```

答案是3.0000。为什么是3.0000?因为x和y都是整型,而两个整型相除,结果是一个舍入的整数,在这个例子中就是3。

如果希望两个整数相除的结果是浮点数,应该先把整数保存到float变量中,但这样稍微有点麻烦,可以使用类型转换临时转换数值的类型:

```
int x = 7;
int y = 2;
float z = (float)x / (float)y;
printf("z = %f\n", z);
```

(float)会把int值转换为float值,计算时就可以把变量当成浮点数来用。事实上,如果编译器发现有整数在加、减、乘、除浮点数,会自动替你完成转换,因此可以减少代码中显式类型转换的次数:

```
float z = (float)x / y;
```

编译器会自动把y的类型转换成float。

你可以在数据类型前加几个关键字,来改变数值的意义:

unsigned

用unsigned修饰的数值只能是非负数。由于无需记录负数,无符号数有更多的位可以使用,因此它可以保存更大的数。unsigned int可以保存0到最大值的数。这个最大值是int可以保存最大值的两倍左右。还有signed关键字,但你几乎从没见过,因为所有数据类型默认都是有符号的。

```
unsigned char c;
```

保存0到255的数。

long

没错,你可以在数据类型前加long,让它变长。long int是加长版的int;long int可以保存范围更广的数字;long long比long更长;还可以对浮点数用long。

```
long double d;
```

极精确的数字。

只有C99和C11标准支持long long。

下面这个程序帮助Head First餐厅的服务员更好地进行服务。代码自动计算总账，并为每笔消费收取消费税，你能填满所有空格吗？

注意：程序将使用多种数据类型，你认为这些数值应该用哪种数据类型？

```
#include <stdio.h>

................ total = 0.0;
................ count = 0;
................ tax_percent = 6;

................ add_with_tax(float f);
{
  ................ tax_rate = 1 + tax_percent / 100 ................ ;
  total = total + (f * tax_rate);
  count = count + 1;
  return total;
}

int main()
{
  ................ val;
  printf("Price of item: ");
  while (scanf("%f", &val) == 1) {
    printf("Total so far: %.2f\n", add_with_tax(val));
    printf("Price of item: ");
  }
  printf("\nFinal total: %.2f\n", total);
  printf("Number of items: %hi\n", count);
  return 0;
}
```

%.2f把浮点数格式化为小数点后两位。

%hi用来格式化short。

练习解答

下面这个程序帮助Head First餐厅的服务员更好地进行服务。代码自动计算总账，并为每笔消费收取消费税，请填满所有空格。

注意：程序将使用多种数据类型，你认为这些数字应该用哪种数据类型？

```
#include <stdio.h>

float total = 0.0;
short count = 0;
short tax_percent = 6;

float add_with_tax(float f);
{
  float tax_rate = 1 + tax_percent / 100.0;
  total = total + (f * tax_rate);
  count = count + 1;
  return total;
}

int main()
{
  float val;
  printf("Price of item: ");
  while (scanf("%f", &val) == 1) {
    printf("Total so far: %.2f\n", add_with_tax(val));
    printf("Price of item: ");
  }
  printf("\nFinal total: %.2f\n", total);
  printf("Number of items: %hi\n", count);
  return 0;
}
```

需要用普通浮点数来表示总额。

一笔订单的项目不会太多，所以我们选择了short。

我们将返回一小笔金额，因此使用float。

分数用float表示就行了。

有了.0，计算就会以浮点数进行，否则表达式会返回整数。

1+ tax_percent/100; 将返回1，因为整型运算6/100 == 0。

价格正好用float表示。

聚焦数据类型大小

不同平台上数据类型的大小不同，那怎么知道int或double占了多少字节？好在C标准库中有几个头文件提供了这些细节。下面这个程序将告诉你int与float的大小。

```
#include <stdio.h>
#include <limits.h>   <- 含有表示整型（比如int和char）大小的值。
#include <float.h>    <- 含有表示float和double类型大小的值。

int main()
{
  printf("The value of INT_MAX is %i\n", INT_MAX);
  printf("The value of INT_MIN is %i\n", INT_MIN);
  printf("An int takes %z bytes\n", sizeof(int));

  printf("The value of FLT_MAX is %f\n", FLT_MAX);
  printf("The value of FLT_MIN is %.50f\n", FLT_MIN);
  printf("A float takes %z bytes\n", sizeof(float));

  return 0;
}
```

最大值

最小值

sizeof返回数据类型占了多少字节。

编译并运行代码，会看到这样的结果：

```
File Edit Window Help HowBigIsBig
The value of INT_MAX is 2147483647
The value of INT_MIN is -2147483648
An int takes 4 bytes
The value of FLT_MAX is 340282346638528859811704183484516925440.000000
The value of FLT_MIN is 0.000000000000000000000000000000000000011754943508222
A float takes 4 bytes
```

不同计算机上的结果很可能不同。

如果你想知道char、double或long的细节呢？也很简单，只要把INT和FLT替换成CHAR（char）、DBL（double）、SHRT（short）或LNG（long）即可。

这里没有蠢问题

问： 为什么不同操作系统的数据类型大小不同？设成一样不是更明了？

答： 为了适应硬件，C语言在不同的操作系统与处理器上使用不同的数据类型大小。

问： 怎么说？

答： C语言诞生之初还是8位机的天下，但现在大部分计算机都是32位和64位的，因为C语言没有指定数据类型的具体大小，所以才能与时俱进。即使新的计算机出来，C语言还是能够很好地适应。

问： 8位、64位到底是什么意思？

答： 从技术上讲，计算机的位数有多种含义，它既可以代表CPU指令的长度，也可以代表CPU一次从存储器读取数据的大小。实际上，位数是计算机能够处理的数值长度。

问： 这和`int`、`double`的大小有什么关系？

答： 如果一台计算机能处理32位的数值，就会把基本数据类型（例如`int`）的大小设为32位。

问： 我知道`int`这样的整数是怎么工作的，但`float`或`double`是怎么保存的呢？计算机如何表示有小数点的数字呢？

答： 一言难尽，大部分计算机使用了IEEE发布的标准（http://tinyurl.com/6defkv6）。

问： 我需要理解浮点数的工作原理吗？

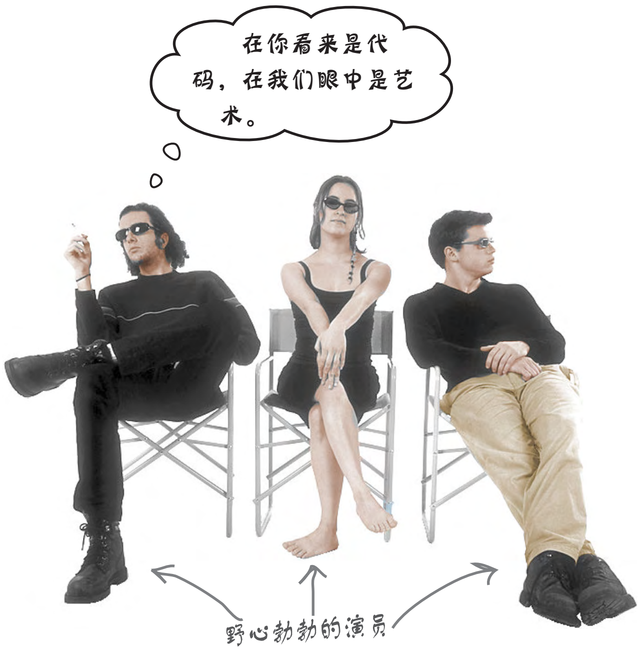
答： 不需要，大部分程序员使用`float`与`double`时不会关注它们的细节。

不好啦，兼职演员来了……

要我说，有些人生来不是写代码的料。有些演员不安于现状，没戏拍的时候替人修改代码，赚点外快，他们决定花时间将算账程序的代码粉饰一新。

当他们改完代码，深深地陶醉在自己的杰作之中……但是有一个小问题。

代码再也无法编译了。



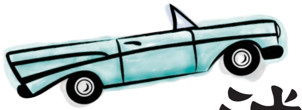
代码到底怎么了

下面是修改后的代码，他们的确改了不少东西。

```
#include <stdio.h>

float total = 0.0;
short count = 0;
/* 6%,比我的经纪人拿的少多了…… */
short tax_percent = 6;

int main()
{
  /* 嘿,我将和梁朝伟联袂出演一部电影 */
  float val;
  printf("Price of item: ");
  while (scanf("%f", &val) == 1) {
    printf("Total so far: %.2f\n", add_with_tax(val));
    printf("Price of item: ");
  }
  printf("\nFinal total: %.2f\n", total);
  printf("Number of items: %hi\n", count);
  return 0;
}

float add_with_tax(float f)
{
  float tax_rate = 1 + tax_percent / 100.0;
  /* 小费呢?口头传授也是要收费的  */
  total = total + (f * tax_rate);
  count = count + 1;
  return total;
}
```

他们先是加了一些注释，然后**改变了两个函数的先后顺序**，仅此而已。

不应该有问题吧，代码应该可以正常工作，你说呢？一切都很顺利，直到他们编译了代码……



试驾

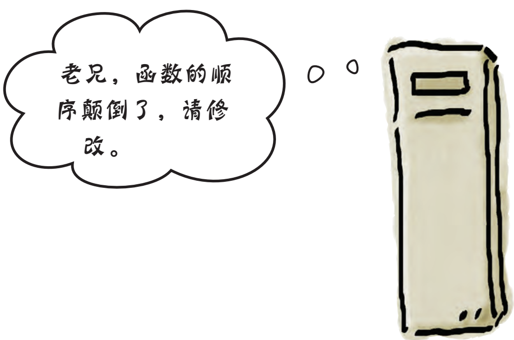
打开控制台编译程序，下面的情况发生了：

```
File Edit Window Help StickToActing
> gcc totaller.c -o totaller && ./totaller
totaller.c: In function "main":
totaller.c:14: warning: format "%.2f" expects type
"double", but argument 2 has type "int"
totaller.c: At top level:
totaller.c:23: error: conflicting types for "add_with_tax"
totaller.c:14: error: previous implicit declaration of
"add_with_tax" was here
```

糟糕。

不妙。`error: conflicting types for 'add_with_tax'`是什么意思？什么是previous implicit declaration？为什么编译器认为`add_with_tax`是`int`？它不应该返回浮点数么？

编译器会忽视注释，加不加都一样，所以问题是由改变函数的顺序所引起的。既然是顺序问题，为什么编译器不返回一条这样的消息：



说真的，为什么这时编译器不帮帮我们？

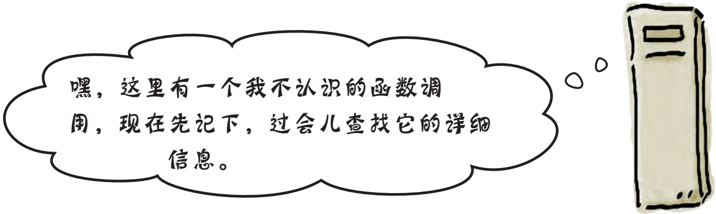
为了理解到底发生了什么，需要进入编译器的灵魂深处，站在它的立场看问题。你会发现编译器不但想帮忙，而且帮过了头。

编译器不喜欢惊喜

当编译器看到这行代码时会发生什么？

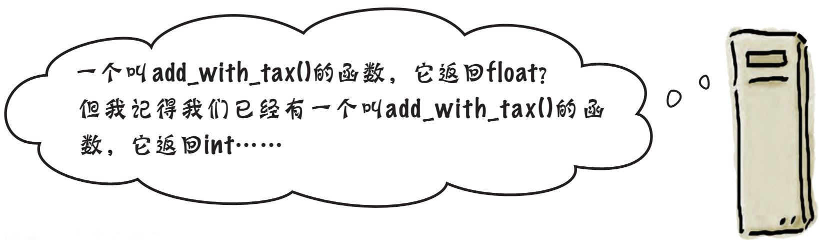
```
printf("Total so far: %.2f\n", add_with_tax(val));
```

1. **编译器看到了一个不认识的函数调用。**
   编译器没有报错，而是认为它会在之后的源文件中找到这个函数的详细信息。编译器先记下，随后会在文件中查找函数，很不幸，问题就出在这个地方……



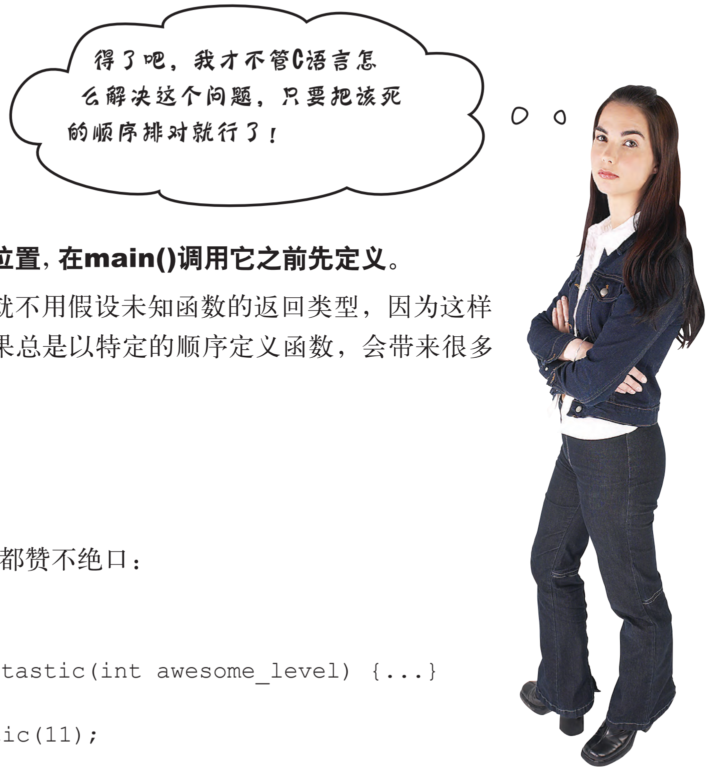
2. **编译器需要知道函数的返回类型。**
   当然了，编译器现在还不知道函数的返回类型，所以只好假设它返回`int`。

3. **等编译器看到实际的函数，返回了“conflicting types for ‘add_with_tax’”错误。**
   因为编译器认为有两个同名的函数，一个是文件中的函数，一个是编译器假设返回`int`的那个。



脑力风暴

计算机假设函数返回`int`，但实际返回了`float`。如果让你设计C语言，你会怎么解决这个问题？



你只要把函数放回正确的位置，在main()调用它之前先定义。

改变函数的顺序，编译器就不用假设未知函数的返回类型，因为这样的假设通常很危险。但如果总是以特定的顺序定义函数，会带来很多后遗症。

调整函数的顺序很痛苦

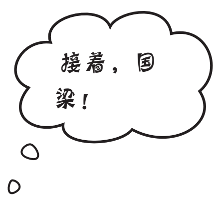
假设你在代码中添加一个新函数，人人都赞不绝口：

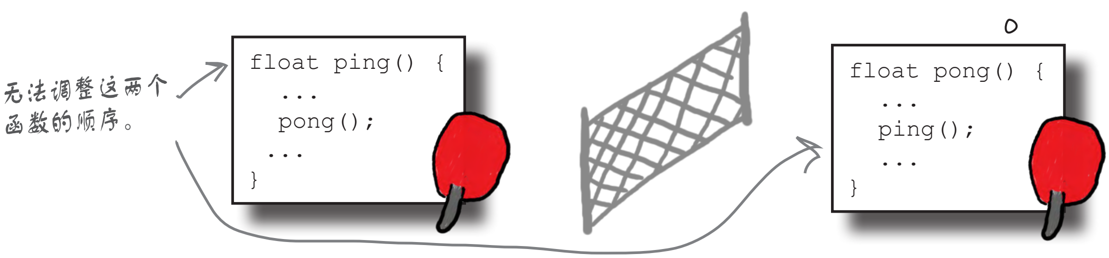
```
int do_whatever(){...}
float do_something_fantastic(int awesome_level) {...}
int do_stuff() {
  do_something_fantastic(11);
}
```

为了完善程序，你决定在do_whatever()函数中调用do_something_fantastic()，为此必须将do_something_ fantastic()函数提前。程序员希望不断改进程序的功能，因此最好有一个两全其美的方法，既不用交换代码的顺序，又能让编译器高兴。

在某些场景中，没有正确的顺序

这种场景有些罕见，但有时你也许想写一些**相互递归**调用的代码：





如果有两个函数，它们互相调用对方，那么总有一个函数在定义前被调用。

出于以上两个原因，你希望更自由地定义函数，那么该怎么做呢？

声明与定义分离

如果编译器一开始就知道函数的返回类型，就不用稍后再找了。为了防止编译器假设函数的返回类型，你可以显式地告诉它。告诉编译器函数会返回什么类型的语句就叫**函数声明**。

声明告诉编译器函数会返回什么类型。

```
float add_with_tax()(float f);
```

声明没有函数体。

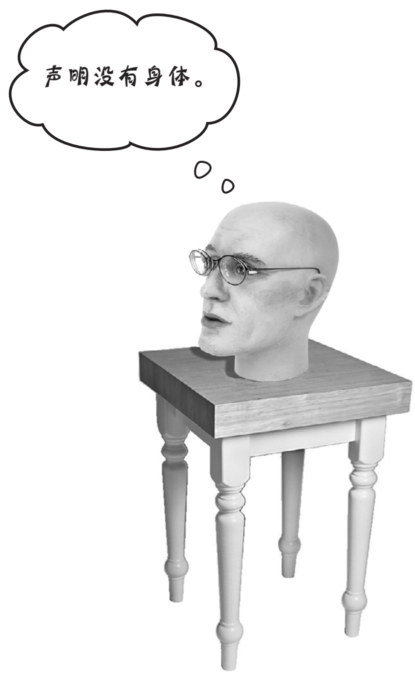
仅以;（分号）结束。

声明只是一个函数**签名**：一条包含函数名、形参类型与返回类型的记录。

一旦声明了函数，编译器就不需要假设，完全可以先调用函数，再定义函数。

如果代码中有很多函数，你又不想管它们在文件中的顺序，可以在代码的开头列出函数声明：

```
float do_something_fantastic();
double awesomeness_2_dot_0();
int stinky_pete();
char make_maguerita(int count);
```



甚至可以把这些声明拿到代码外，放到一个头文件中。你已经用**头文件**包含过C标准库中的代码：

```
#include <stdio.h>
```

包含一个叫stdio.h的头文件中的内容。

去看看如何创建你自己的头文件。

创建第一个头文件

为了创建头文件，只要做两件事：

1 创建一个扩展名为.h的文件。

如果程序叫`totaller`，就创建一个叫*totaller.h*的文件，并把你的声明写在里面：

```
float add_with_tax(float f);
```

totaller.h

不用在头文件中包含`main()`函数，反正也没有函数会调用它。

2 在主代码中包含头文件。

应该在代码的顶部加一句`include`：

在其他include行后加上这句include。

```
#include <stdio.h>
#include "totaller.h"
...
```

totaller.c

头文件的名字用双引号括起来，而不是尖括号，它们的区别是什么？当编译器看到尖括号，就会到标准库代码所在目录查找头文件，但现在你的头文件和*.c*文件在同一目录下，用引号把文件名括起来，编译器就会在本地查找文件。

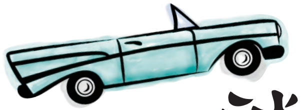
本地头文件也可以带目录名，但通常会把它和C文件放在相同目录中。

当编译器在代码中读到`#include`，就会读取头文件中的内容，仿佛它们本来就在代码中。

把声明放到一个独立的头文件中有两大优点，第一是主代码变短了，第二个优点你会在几页之后看到。

先来看看头文件能否解决眼前的问题。

#include是预处理命令。



试驾

当编译代码时，结果如下：

这次没有错误消息。

```
File Edit Window Help UseHeaders
> gcc totaller.c -o totaller
```

编译器从头文件中读取到了函数声明，因此不必猜测函数返回什么类型，函数的顺序也就无关紧要了。

为了确保一切正常，你可以运行生成的程序，看它能否像刚才一样工作。

```
File Edit Window Help UseHeaders
> ./totaller
Price of item: 1.23
Total so far: 1.30
Price of item: 4.57
Total so far: 6.15
Price of item: 11.92
Total so far: 18.78
Price of item: ^D
Final total: 18.78
Number of items: 3
```

按Crtl-D停止程序，不再输入单价。

变身编译器

下面这个程序的一部分丢失了，你的工作是扮演编译器，在右侧的候选代码块分别填入空白处后，你会做什么？

候选代码从这里开始。

```
#include <stdio.h>

[                                        ]

  printf("水星上一天有 %f 小时\n", day);
  return 0;
}

float mercury_day_in_earth_days()
{
  return 58.65;
}

int hours_in_an_earth_day()
{
  return 24;
}
```

这些是代码片段。

选择你认为正确的说法，在方框里打勾。

```
float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
  float length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

- ☐ 编译通过。
- ☐ 显示一条警告。
- ☐ 程序将正确运行。

```
float mercury_day_in_earth_days();

int main()
{
  float length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

- ☐ 编译通过。
- ☐ 显示一条警告。
- ☐ 程序将正确运行。

```
int main()
{
  float length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

- ☐ 编译通过。
- ☐ 显示一条警告。
- ☐ 程序将正确运行。

```
float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
  int length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

- ☐ 编译通过。
- ☐ 显示一条警告。
- ☐ 程序将正确运行。

变身编译器解答

下面这个程序的一部分丢失了，你的工作是扮演编译器，在右侧的候选代码块分别填入空白处后，你会做什么？

```
#include <stdio.h>

  printf("水星上一天有 %f 小时\n", day);
  return 0;
}

float mercury_day_in_earth_days()
{
  return 58.65;
}

int hours_in_an_earth_day()
{
  return 24;
}
```

```
float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
  float length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

- [x] 编译通过。
- [ ] 显示一条警告。
- [x] 程序将正确运行。

会有一个警告，因为在调用hours_in_an_earth_day()函数前没有声明它，但程序仍能正确运行，因为编译器猜函数会返回int。①

```
float mercury_day_in_earth_days();

int main()
{
  float length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

- [x] 编译通过。
- [x] 显示一条警告。
- [x] 程序将正确运行。

程序无法编译，因为在调用返回值为float类型的函数前没有声明它。

```
int main()
{
  float length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

- [ ] 编译通过。
- [x] 显示一条警告。
- [ ] 程序将正确运行。

程序可以编译，而且没有警告，但不能正确运行，因为有一个舍入的问题。

length_of_day变量应该是float。

```
float mercury_day_in_earth_days();
int hours_in_an_earth_day();

int main()
{
  int length_of_day = mercury_day_in_earth_days();
  int hours = hours_in_an_earth_day();
  float day = length_of_day * hours;
```

- [x] 编译通过。
- [ ] 显示一条警告。
- [ ] 程序将正确运行。

① 只有使用gcc编译器的-std=c99选项编译后才会有该警告。——译者注

这里没有蠢问题

问：也就是说int函数可以没有声明？

答： 不一定，除非你想共享代码，马上你就会看到。

问：我有点纳闷，你提到了编译器预处理，为什么编译器需要预处理？

答： 严格意义上讲，编译器只完成编译的步骤，即把C源代码转化为汇编语言。但宽泛地讲，编译是将C源代码转化为可执行文件的整个过程，这个过程由很多阶段组成，而gcc允许你控制这些阶段。gcc会预处理和编译代码。

问：什么是预处理？

答： 预处理是把C源代码转化为可执行文件的第一个阶段。预处理会在正式编译开始之前修改代码，创建一个新的源文件。拿你的代码来说，预处理会读取头文件中的内容，插入主文件。

问：预处理器会真的创建一个文件吗？

答： 不会，为了提高编译的效率，编译器通常会用管道在两个阶段之间发送数据。

问：为什么有的头文件用引号括起来，有的用尖括号？

答： 严格来说，这是由编译器的工作方式决定的。通常情况下，引号表示以相对路径查找头文件，如果不加目录名，只包含一个文件名，编译器就会在当前目录下查找头文件；如果用了尖括号，编译器就会以绝对路径查找头文件。

问：编译器在寻找头文件时会查找哪些目录？

答： gcc知道标准库的头文件被保存在哪里，在类Unix操作系统中，头文件通常保存在*/usr/local/include*、*/usr/include*这些地方。

问：gcc就是这样找到*stdio.h*的，是吗？

答： 是的，在类Unix操作系统中，*stdio.h*位于*/usr/include/stdio.h*；如果在Windows中安装了MinGW编译器，*stdio.h*就很有可能在*C:\MinGW\include\stdio.h*中。

问：我可以创建我自己的库吗？

答： 可以，你会在后面几章中学到。

要点

- 如果编译器发现你调用了一个它没见过的函数，就会假设这个函数返回int。
- 所以如果想在定义函数前就调用它，就可能出问题。
- 函数声明在定义函数前就告诉编译器函数长什么样子。
- 如果在源代码顶端声明了函数，编译器就知道函数返回什么类型。
- 函数声明通常放在头文件中。
- 可以用#include让编译器从头文件中读取内容。
- 编译器会把包含进来的代码看成源文件的一部分。

先生，这个位子有人订了……

C语言是一种很小的语言，所有的保留字都在这里（排名不分先后）。

所有的C程序都可以分解为这些关键字和若干符号。如果用这些关键字来命名，编译器会坐立不安。

auto	if	break
int	case	long
char	register	continue
return	default	short
do	sizeof	double
static	else	struct
entry	switch	extern
typedef	float	union
for	unsigned	goto
while	enum	void
const	signed	volatile

如果有共同特性……

你用C语言写了几个程序以后，就想从其他程序中复用某些函数或特性。打个比方，看右边那两份程序说明书。

file_hider

从文件中读取内容，使用异或加密创建一个加密版本。

message_hider

从标准输入读取字符串，进行异或加密，然后在标准输出显示密文。

异或加密是一种很简单的加密方法，它将文本中的每个字符与某个值进行异或。虽然异或加密很不安全，但易于操作，而且加、解密使用同一段代码。下面就是一段加密文本的代码：

```
void encrypt(char *message)
{
  char c;
  while (*message) {
    *message = *message ^ 31;
    message++;
  }
}
```

void表示什么都不返回。

传一个数组指针给函数。

循环遍历数组，加密每个字符。

表示将把message的每个字符异或上数字31。

可以对char进行运算，因为它是数值类型。

……最好可以共享代码

显然，这两个程序会使用相同的encrypt()函数，你大可把代码从一个程序复制到另一个，对吧？但这只适用于代码少的情况，如果有很多代码该怎么办？再说万一以后我们要修改encrypt()函数，就要改两个地方。

为了让代码可以良好地扩展，需要想办法复用相同代码，例如在多个程序之间共享函数。

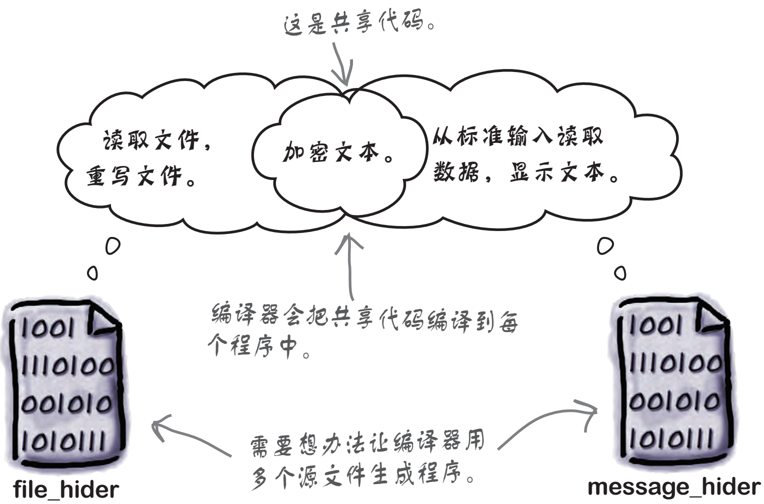
怎样才能共享函数？

脑力风暴

你想在程序间共享函数。假如是你发明了C语言，你会怎么做？

把代码分成多个文件

如果想让多个文件共享一组代码，自然要把共享的代码放在一个单独的*.c*文件中。只要编译器在编译程序时包含共享代码，就可以在多个程序中使用相同的代码了。一旦你想修改共享代码，只要修改一处就行了。



如果你想把共享代码放在一个单独的*.c*文件中，会有一个问题。到目前为止，每次创建程序时都只用了一个*.c*源文件。如果程序叫`blitz_hack`，就会用一个叫*blitz_hack.c*的源文件。

但现在你想给编译器一组源文件，然后说：“用这些文件创建程序。”你要怎么做呢？应该用什么`gcc`语法？更重要的是，对编译器来说，用多个文件创建一个可执行程序是什么意思？编译器怎么工作？它又如何衔接这些代码？

为了理解C编译器如何用多个文件创建程序，我们一起去看看编译的过程……

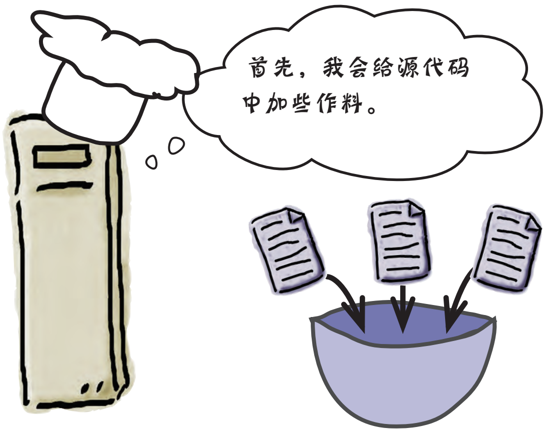
编译的幕后花絮

为了理解编译器怎么把多个源文件编译成一个程序，需要拉开帷幕，看看编译器到底怎么工作。

1 预处理：修改代码。
编译器要做的第一件事就是修改代码。编译器需要用#include指令添加相关头文件；编译器可能还需要跳过程序中的某些代码，或补充一些代码。改完以后就可以随时编译源代码了。

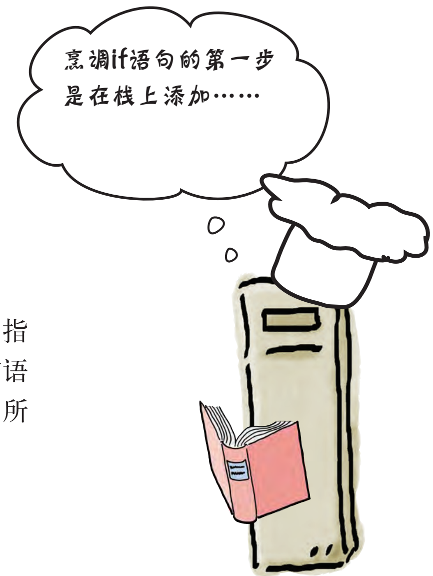
“指令”是“命令”更花哨的说法。

可以用#define和#ifdef来实现，稍后将学习如何使用它们。



2 编译：转换成汇编代码。
C语言看似底层，但计算机还是无法理解它。计算机只理解更低层的机器代码指令。而生成机器代码的第一步就是把C语言源代码转化为汇编语言代码，看起来像这样：

```
movq   -24(%rbp), %rax
movzbl  (%rax), %eax
movl   %eax, %edx
```



是不是很难懂？汇编语言描述了程序运行时中央处理器需要执行的指令。C编译器有很多食谱，覆盖了C语言的方方面面，比如如何把if语句或函数调用转化为一串汇编语言指令。但汇编语言还不够底层，所以我们需要……

3 **汇编：生成目标代码。**
编译器需要将这些符号代码汇编成机器代码或目标代码，即CPU内部电路执行的二进制代码。

不骗你，这是一段用机器代码编的笑段子。

```
10010101 00100101 11010101 01011100
```

你已经把C语言源代码转化成电路所需的0和1了。还差最后一步，你给了编译器几个文件来编译程序，编译器会为每个源文件生成一个目标代码，为了生成可执行程序，还需要对这些目标文件做一件事……

4 **链接：放在一起。**
一旦有了全部的目标代码，就需要像拼“七巧板”那样把它们拼在一起，构成可执行程序。当某个目标代码的代码调用了另一个目标代码的函数时，编译器会把它们连接在一起。同时，链接还会确保程序能够调用库代码。最后，程序会写到一个可执行程序文件中，文件格式视操作系统而定，操作系统会根据文件格式把程序加载到存储器中运行。

怎么让gcc知道我们想用几个单独的源文件生成可执行程序呢?

共享代码需要自己的头文件

如果想在多个程序之间共享*encrypt.c*代码，需要想办法让这些程序知道它，为此你可以用头文件。

你将在encrypt.c中包含这个头文件。

```
void encrypt(char *message);
```

encrypt.h

```
#include "encrypt.h"

void encrypt(char *message)
{
  char c;
  while (*message) {
    *message = *message ^ 31;
    message++;
  }
}
```

encrypt.c

在程序中包含encrypt.h

在这里使用头文件不是为了能够调整函数之间的顺序，而是为了让其他程序知道encrypt()函数：

将包含encrypt.h，这样程序就有了encrypt()函数的声明。

```
#include <stdio.h>
#include "encrypt.h"

int main()
{
  char msg[80];
  while (fgets(msg, 80, stdin)) {
    encrypt(msg);
    printf("%s", msg);
  }
}
```

message_hider.c

主程序有*encrypt.h*，这表示编译器知道encrypt()函数，这样才能编译代码。在链接阶段，编译器会把*message_hider.c*中的encrypt(msg)调用连接到*encrypt.c*中的encrypt()函数。

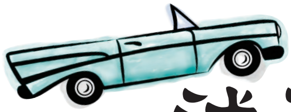
最后，为了把所有东西编译到一起，只需把源文件传给gcc：

```
gcc message_hider.c encrypt.c -o message_hider
```

共享变量

你已经知道如何在不同的文件之间共享函数，但如果你想共享变量呢？为了防止两个源文件中的同名变量相互干扰，变量的作用域仅限于某个文件内。如果你想共享变量，就应该在头文件中声明，并在变量名前加上**extern**关键字：

```
extern int passcode;
```



试驾

看看你编译message_hider程序时会发生什么：

需要用两个源文件编译代码。

当运行程序时，你可以输入文本，然后看到加密文本。

甚至可以把encrypt.h的内容传给程序加密。

message_hider程序使用了encrypt.c中的encrypt()函数。

```
File Edit Window Help Shhh...
> gcc message_hider.c encrypt.c -o message_hider
> ./message_hider
I am a secret message
V?~r?~?lz|mzk?rzll~xz
> ./message_hider < encrypt.h
ipv{?zq|mfok7|w~m5?rzll~xz6$
>
```

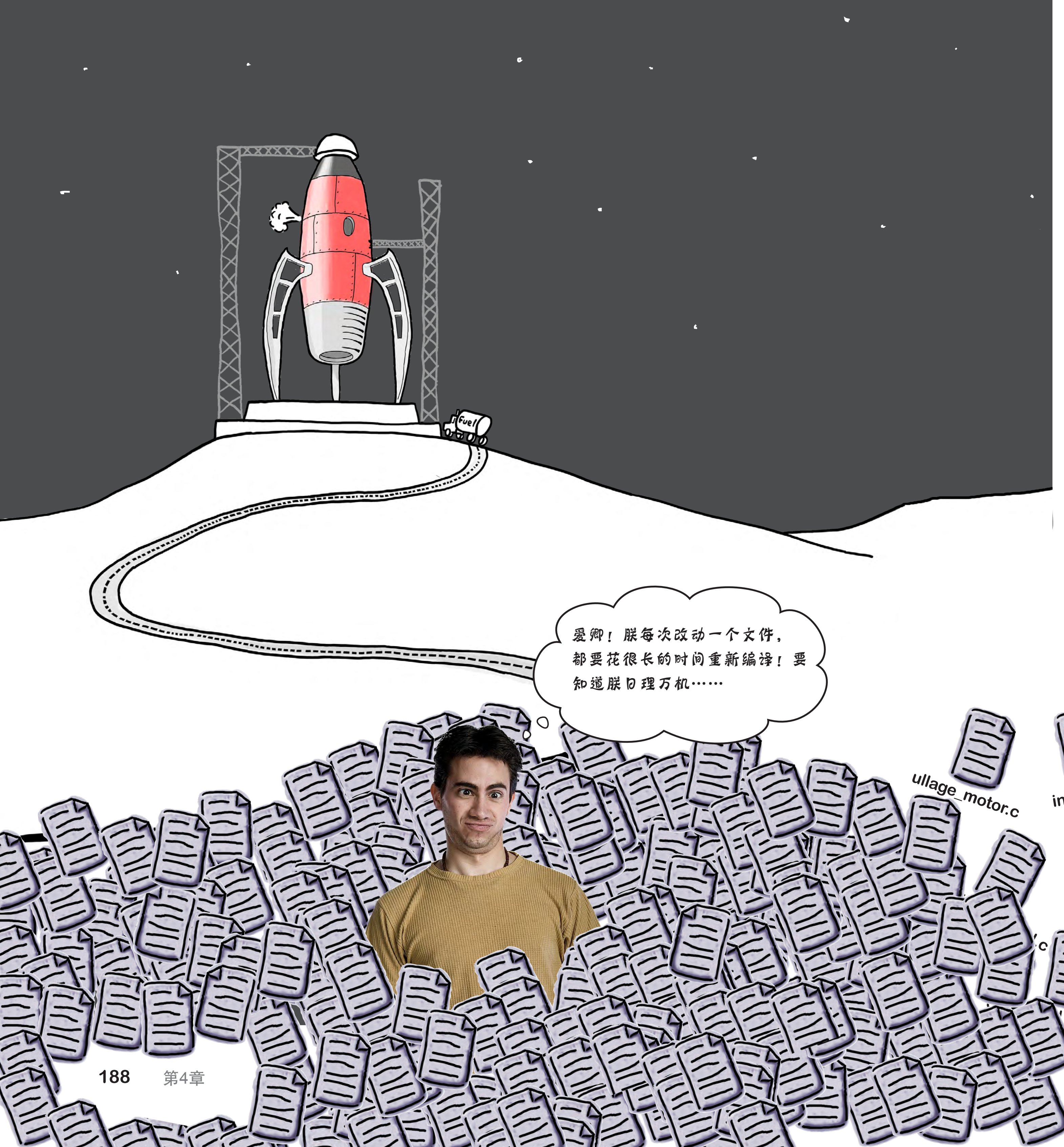
程序正确运行了。只要把encrypt()函数放在一个单独的文件中，就可以在任何程序中使用它了。假如你想修改encrypt()函数，把它变得更安全，只要修改*encrypt.c*文件就行了。

要点

- 为了共享代码，可以把代码放到一个单独的C文件中。
- 需要把函数声明放到一个单独的.h头文件中。
- 在所有需要使用共享代码的C文件中包含这个头文件。
- 在编译的命令中列出所有C文件。

滑野雪

赶快用encrypt()写一个程序吧。别忘了，encrypt()还可以用来解密喔。

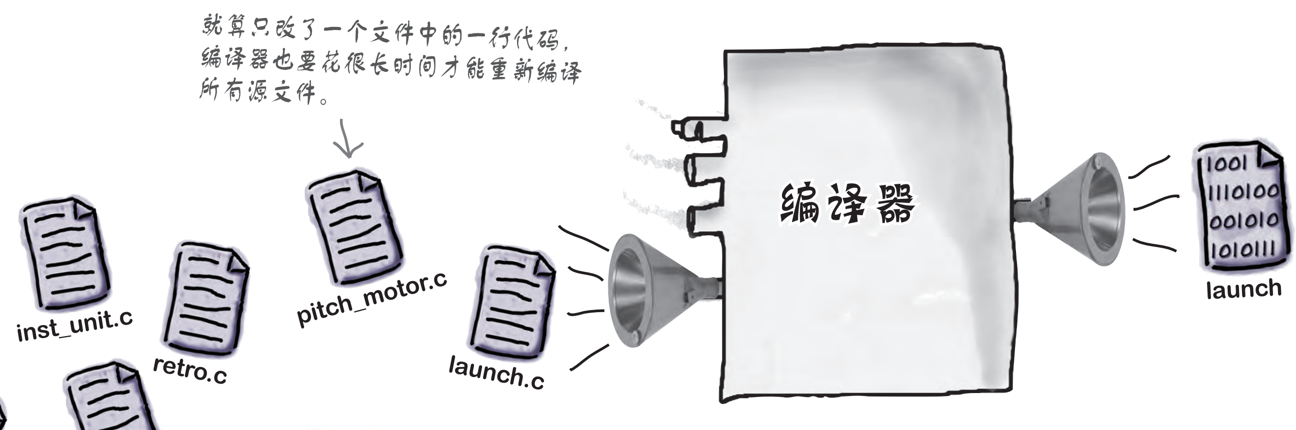

爱卿！朕每次改动一个文件，都要花很长的时间重新编译！要知道朕日理万机……
ullage_motor.c

又不是造火箭……还真是！

把程序分解成独立的源文件，不仅意味着可以在不同程序之间共享代码，还意味着可以开始创建真正的大程序了。为什么？因为现在可以把程序分解成更小的自治代码片段。比起在一个庞大的源文件中搞定一切，现在你可以有很多更简单的文件，它们更容易理解、维护和测试。

使用多个源文件的优点是可以开始创建大程序了。缺点嘛，还是可以开始创建大程序了。C编译器是很高效的软件，它把程序改头换面了好几次：编译器会修改源代码，把许许多多文件链接起来，还不会挤爆存储器，编译器甚至还会优化代码，尽管编译器做了那么多事情，但它还是很快。

但如果用很多文件来创建程序，编译代码的时间就变得很重要。假设一个大型项目要花一分钟编译，听起来可能不长，但足以打断你的思路。当你修改了一行代码，希望尽快看到运行结果，如果每次都要等足一分钟才能看到结果，就会减慢速度。



脑力风暴

仔细想想，就算是一个很小的改动，也要花很长时间编译才能看到结果。凭你对编译过程的了解，怎样才能提高重新编译程序的速度？

不要重新编译所有文件

如果只修改了一两个源文件便为程序重新编译所有源文件就是浪费。当运行下面这条命令时，想想会发生什么？

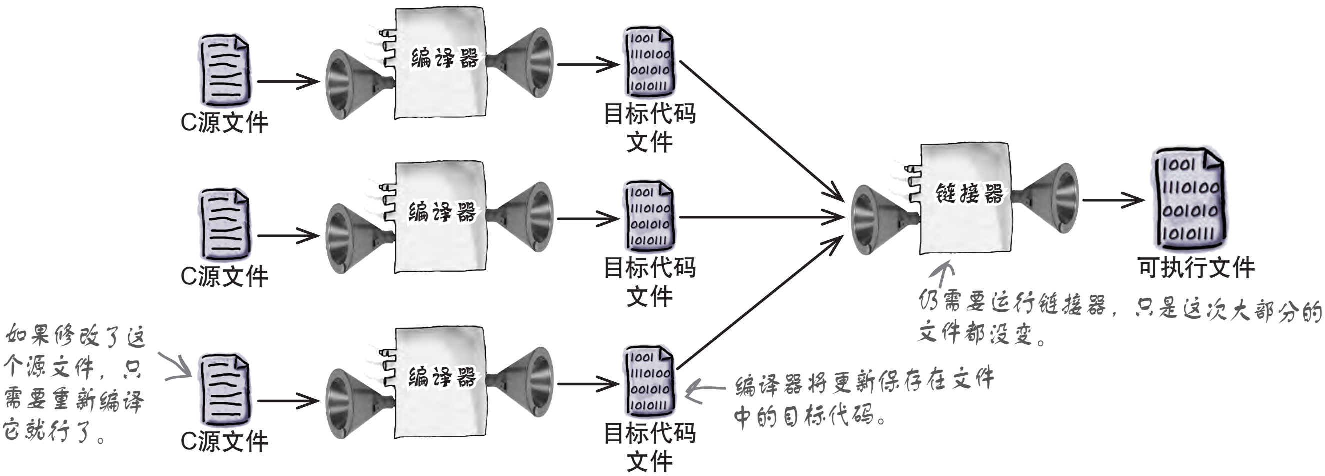
这里跳过了一些文件名。

```
gcc reaction_control.c pitch_motor.c ... engine.c -o launch
```

编译器会做什么？它会对所有源文件以及那些没有改动过的文件分别运行预处理器、编译器和汇编器。既然源代码没有变，它们生成的**目标代码**也不会变，如果编译器每次都会为所有文件生成目标代码，你该做些什么？

保存目标代码的副本

如果让编译器把生成的目标代码保存到文件中，就不需要重新生成它了，除非你修改了源代码。假如你修改了某个源文件，可以重新创建这**一个文件**的目标代码，然后把所有的目标文件传给编译器，让编译器把它们链接起来。



如果你修改了一个文件，就必须用它重新创建目标代码文件，但不需要为其他文件创建目标代码。然后可以把所有目标代码文件传给链接器，创建新的程序。

怎么才能让gcc把目标代码保存在文件中？然后让编译器把目标文件链接起来？

首先，把源代码编译为目标文件

gcc -c会编译代码，但不会链接目标文件。

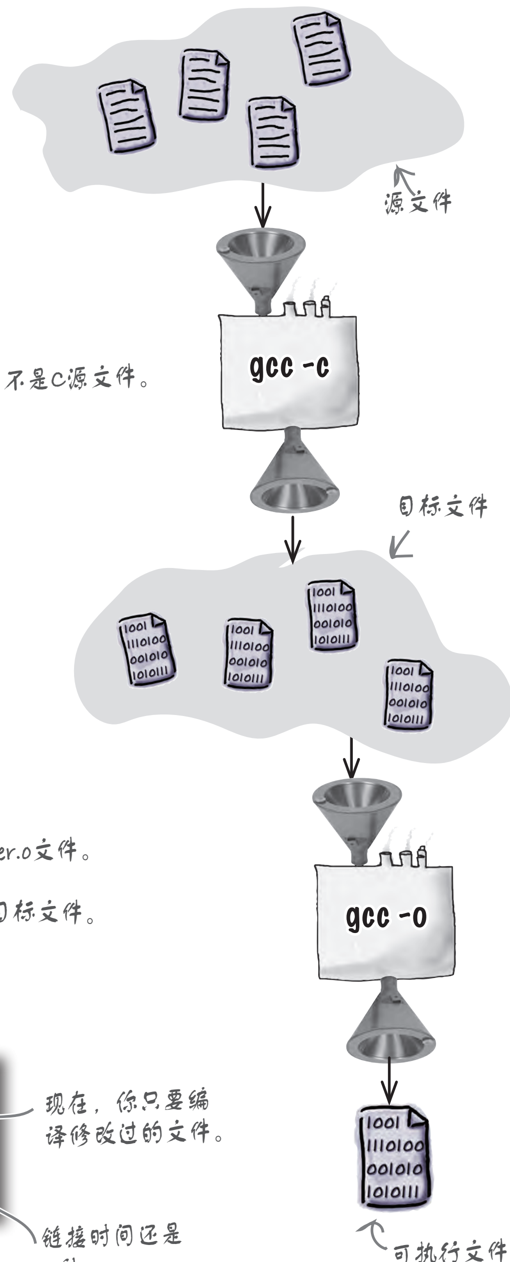
为了得到所有源文件的目标代码，可以输入以下命令：

为所有文件创建目标代码。→

```
gcc -c *.c
```

← 操作系统会把*.c替换成所有的C源文件的文件名。

*.c会匹配当前目录下所有的C源文件，-c告诉编译器你想为所有源文件创建目标文件，但不想把目标文件链接成完整的可执行程序。



然后，把目标文件链接起来

既然你有了一批目标文件，就可以用一条简单的编译命令把它们链接起来。这次要把目标文件的名字给编译器，而不是C源文件的名字。

很像你以前用过的编译命令。→

```
gcc *.o -o launch
```

← 列出目标文件，而不是C源文件。

↑ 匹配当前目录下所有目标文件。

编译器能够识别这些文件是目标文件，而非源文件，因此它会跳过大部分编译步骤，直接把目标文件链接为一个叫launch的可执行程序。

和以前一样，现在你有了一个编译好的程序，同时你也得到了一批目标文件，可以在需要时随时把它们链接起来。如果要修改其中一个文件，只需要重新编译这一个文件，然后重新链接程序即可：

这是唯一修改过的文件。→

```
gcc -c thruster.c
gcc *.o -o launch
```

← 重新创建thruster.o文件。

← 重新链接所有目标文件。

虽然必须输入两条命令，但节省了很多时间。

	以前	现在
编译时间：	2分30秒	2秒
链接时间	6秒	6秒

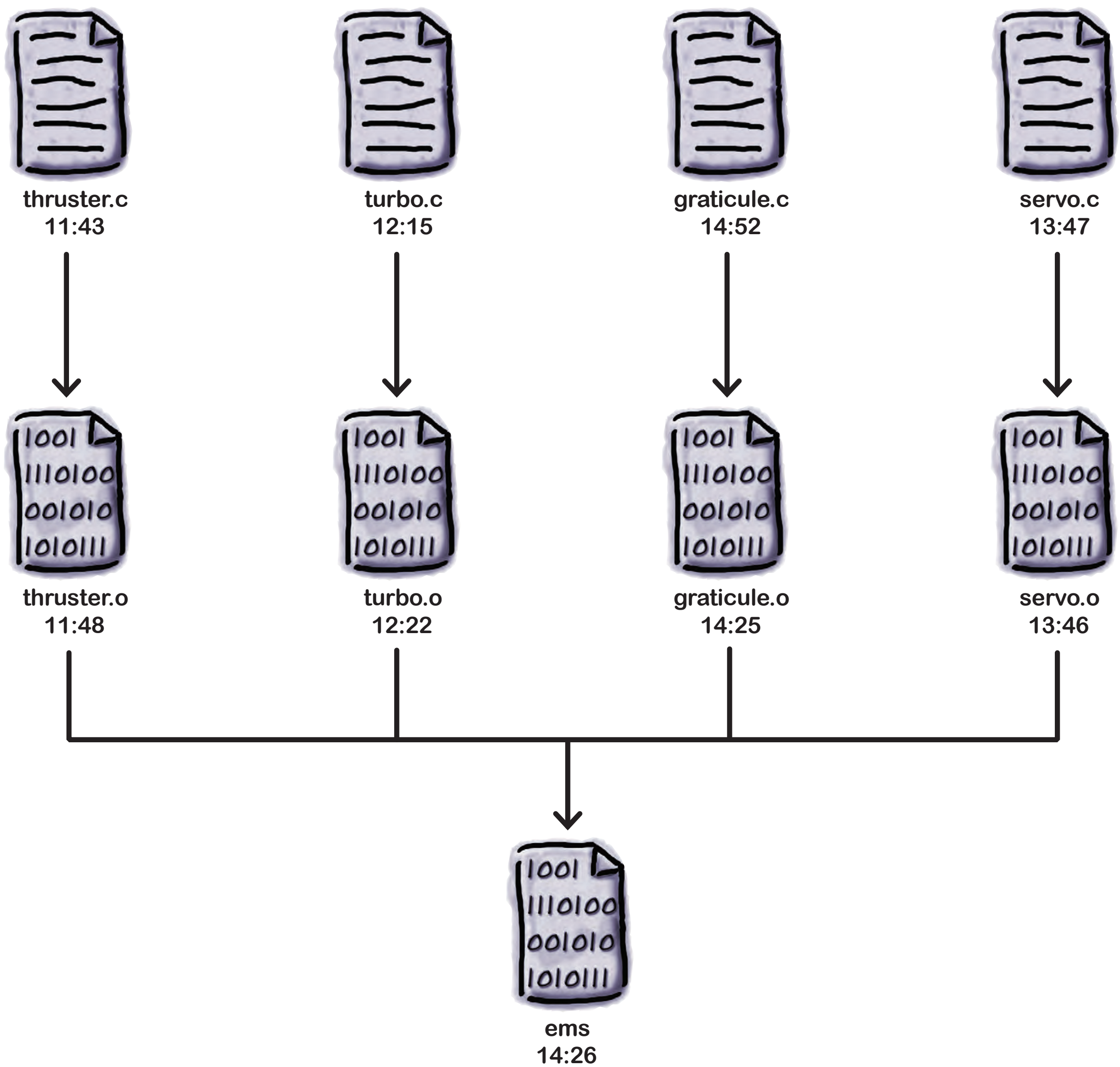
以前，要编译所有文件。

构建提速95%。

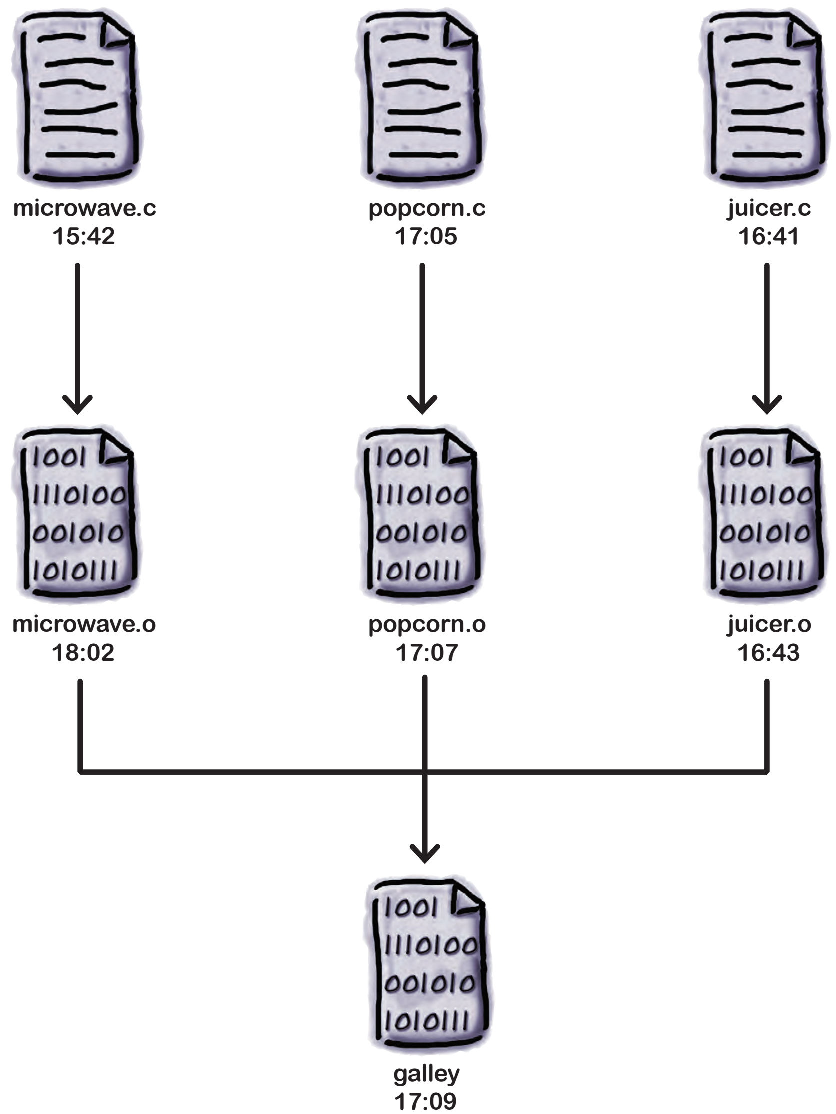
现在，你只要编译修改过的文件。

链接时间还是6秒。

这里的代码用来控制飞船的引擎管理系统（engine management system），每个文件都有一个时间戳。为了得到最新的ems程序，你认为需要重新创建哪些文件？圈出你认为需要更新的文件。

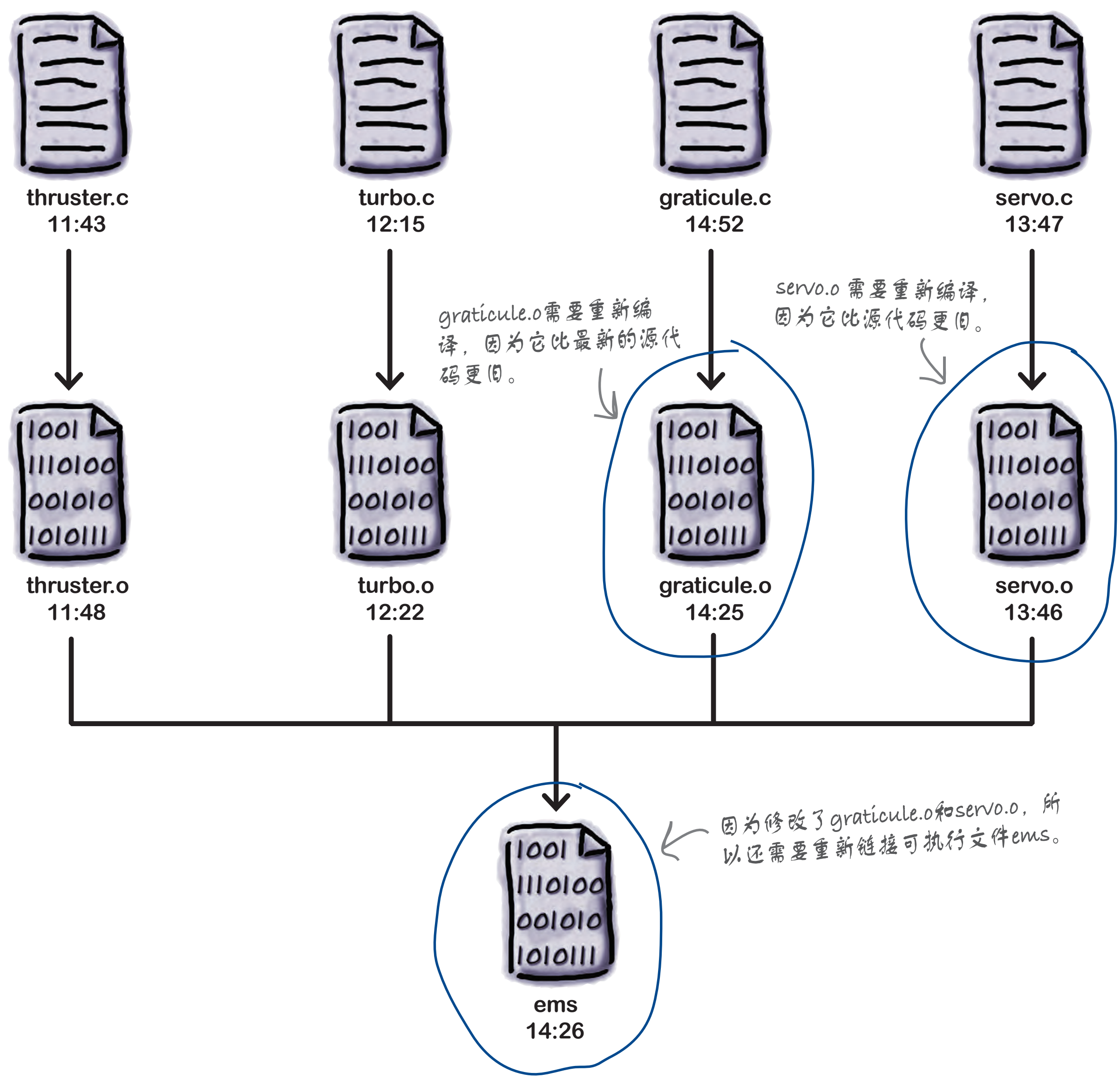


而在厨房中，厨师也需要确保他们的代码是最新的。查看文件的更新时间，哪些文件需要更新？

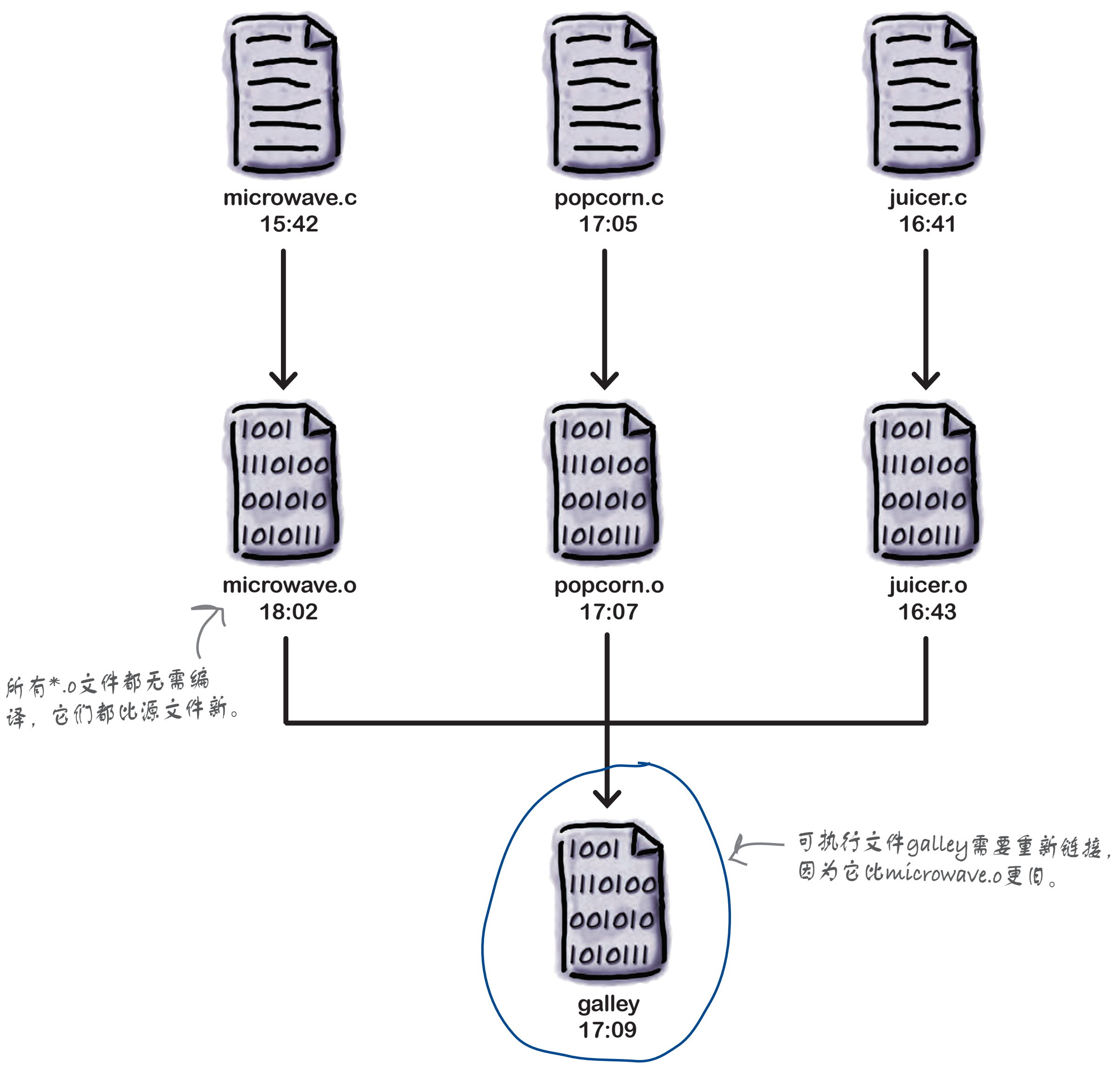


练习解答

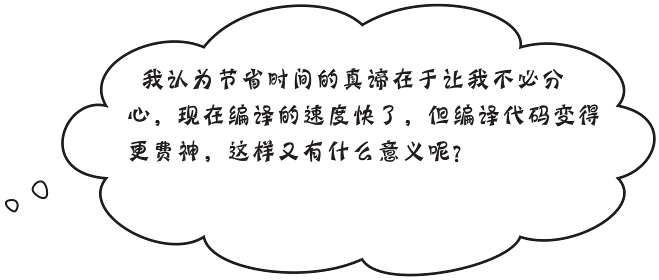
这里的代码用来控制飞船的引擎管理系统（engine management system），每个文件都有一个时间戳，你圈出了需要重新生成的文件，重新生成它们就能得到最新的ems可执行文件。



而在厨房中，厨师也需要确保他们的代码是最新的。查看文件的更新时间，哪些文件需要更新？



记不住修改了哪些文件



没错，局部的编译加快了，但你不得不三思而后行，以确保该编译的文件都编译了。

如果只改了一个源文件，没什么问题，但如果你改了很多文件，就很容易忘记重新编译其中的某些文件，也就是说新程序体现不出所有的变化。当然，在你发布最后的程序之前，完全可以重新编译每个文件，但如果还处于开发阶段，你绝对不想这么做。

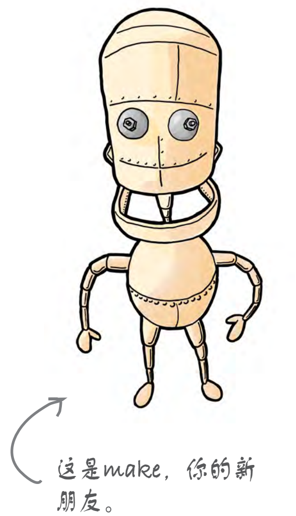
虽说寻找需要编译的文件是一个十分机械化的**过程**，但如果手工来做，很容易遗漏某些修改。

有没有什么方法可以自动化这个过程？

要是有工具能自动重新编译那些修改过的源文件就好了，但我知道这是在痴人说梦……

用make工具自动化构建

只要记下修改过哪些文件，就可以很快地用gcc编译程序。这是一件很麻烦的事，但很容易自动化。想象有一个文件，这个文件是由另一个文件生成的，比如从源文件编译过来的目标文件：



如果thruster.c文件较新，就需要重新编译。 → **thruster.c** ⟶ **thruster.o** ← 如果thruster.o文件较新，就不需要重新编译。

你怎么知道*thruster.o*文件是否需要重新编译呢？只要看一下这两个文件的时间戳就行了，如果*thruster.o*文件比*thruster.c*文件旧，就需要重新创建*thruster.o*；否则就说明*thruster.o*已经是最新的了。

非常简单的规则。如果你掌握了某样东西的简单规则，别多想，**自动化它**……

make是一个可以替你运行编译命令的工具。make会检查源文件和目标文件的时间戳，如果目标文件过期，make就会重新编译它。

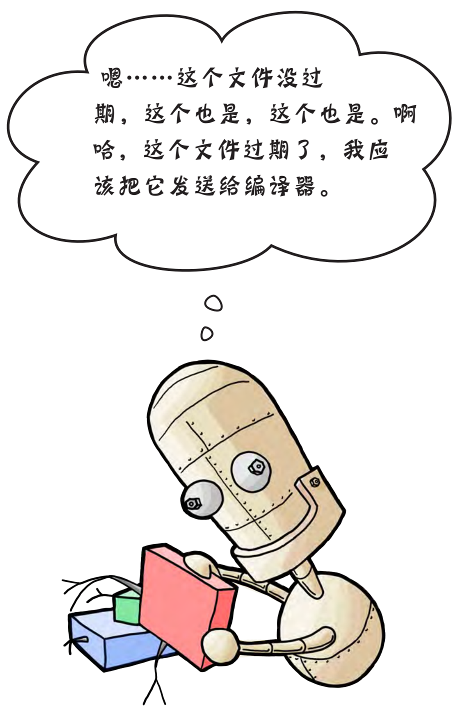
但是做到所有这些事情前，需要告诉make源代码的一些情况。make需要知道文件之间的依赖关系，同时还需要告诉它你具体想如何构建代码。

make需要知道什么？

make编译的文件叫**目标**（target）。严格意义上讲，make不仅仅可以用来编译文件。目标可以是任何用其他文件生成的文件，也就是说目标可以是一批文件压缩而成的压缩文档。

对每个目标，make需要知道两件事：

- **依赖项。**
  生成目标需要用哪些文件。
- **生成方法。**
  生成该文件时要用哪些指令。



依赖项和生成方法合在一起构成了一条**规则**。有了规则，make就知道如何生成目标。

make是如何工作的

假设你想要把*thruster.c*编译成目标代码*thruster.o*，依赖项和生成方法分别是什么？

thruster.c ⟶ thruster.o

*thruster.o*就叫目标，因为你想生成这个文件。*thruster.c*是依赖项，因为编译器在创建*thruster.o*时需要它。那么生成方法呢？生成方法就是将*thruster.c*转化为*thruster.o*的编译命令。

```
gcc -c thruster.c
```

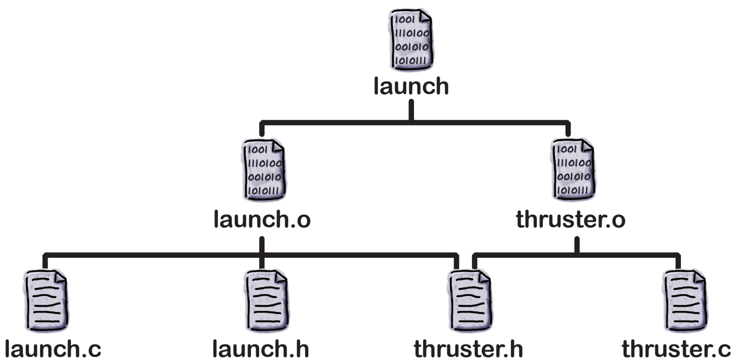
创建thruster.o的规则。

说得通吧？你只要告诉make依赖项以及生成方法，就可以让make决定什么时候重新编译*thruster.o*。

你可以做得更多。一旦创建了*thruster.o*文件，接下来就要用它来创建launch程序，launch文件也可以设为目标，因为你想生成它，launch的依赖项是所有*.o*目标文件，生成方法如下：

```
gcc *.o -o launch
```

一旦make得到了所有的依赖项和生成方法，那么只要让它创建launch程序就行了，make会处理细节。



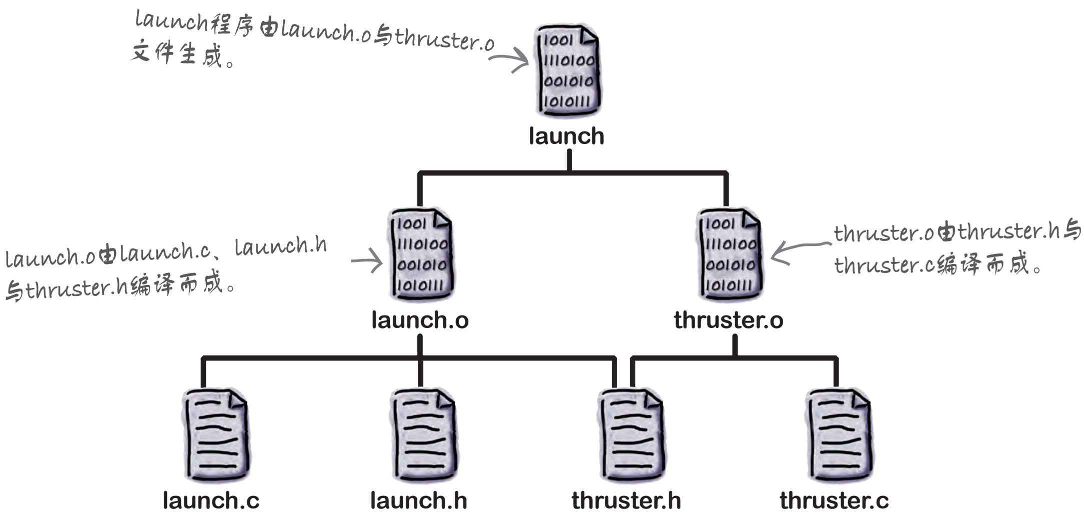
怎么把依赖项和生成方法告诉make？我们来瞧瞧。

make在Windows中另有其名。

来自UNIX世界的make在Windows中有很多“艺名”，MinGW版的make叫mingw32-make，而微软有自己的NMAKE。

用makefile向make描述代码

所有目标、依赖项和生成方法的细节信息需要保存在一个叫*makefile*或*Makefile*的文件中，为了弄明白它是怎么工作的，下面假设要用一对源文件创建`launch`程序：



`launch`程序由*launch.o*和*thruster.o*文件链接而成，这两个文件又是由相应的C文件和头文件编译而成，*launch.o*文件还依赖*thruster.h*文件，因为*thruster.c*需要调用*thruster.h*中的函数。

你将在makefile中这样描述构建过程：

目标是你想生成的文件。

这是目标。

```
launch.o: launch.c launch.h thruster.h
	gcc -c launch.c

thruster.o: thruster.h thruster.c
	gcc -c thruster.c

launch: launch.o thruster.o
	gcc launch.o thruster.o -o launch
```

launch.o依赖这三个文件。

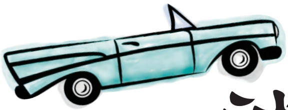
有三条规则。

这是创建thruster.o的生成方法。

生成方法必须以tab开始。

生成方法都必须以tab开头。

如果尝试用空格缩进，就无法生成程序。



试驾

将make规则保存在当前目录下一个叫*Makefile*的文本文件中，然后打开控制台，输入以下命令：

让make创建launch文件。

make首先需要用这行命令创建launch.o。

然后make需要用这行命令创建thruster.o。

最后，make把目标文件链接在一起，创建launch程序。

```
File  Edit  Window  Help  MakeItSo
> make launch
gcc -c launch.c
gcc -c thruster.c
gcc launch.o thruster.o -o launch
```

可以看到make为了创建launch程序执行了一连串的命令。但如果修改了*thruster.c*文件，再运行一次make，会发生什么呢？

make无需编译launch.c。

launch.o已经是最新版本了。

```
File  Edit  Window  Help  MakeItSo
> make launch
gcc -c thruster.c
gcc launch.o thruster.o -o launch
```

make会跳过创建新的*launch.o*这一步，只编译*thruster.o*，然后重新链接程序。

这里没有蠢问题

问： **make很像ant？**

答： 应该说ant和rake[1]这样的构建工具像make才对，make是最早出现的用来从源代码自动构建程序的工具。

问： **编译源代码要做那么多事情，make真的那么管用吗？**

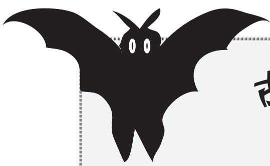
答： 是的，make非常有用。对小项目来说，make可能节约不了太多时间，但一旦有很多文件，手动编译、链接它们会很痛苦。

问： **如果我在Windows上写了一个makefile，在Mac或Linux上能用吗？**

答： makefile会调用底层操作系统的命令，所以有时不能在其他操作系统中使用。

问： **除了编译代码，我能用make做其他事情吗？**

答： 可以，虽然make一般用来编译代码，但你也可以用它充当命令行下的安装程序或源代码控制工具。事实上，任何可以在命令行中执行的任务，你都可以用make来做。



古墓迷案

为什么要用tab缩进？

用空格缩进“生成方法”比用tab缩进方便多了，那么为什么make规定必须使用tab呢？这个嘛，make之父Stuart Feldman曾说过：

“为什么要在第一列中使用tab？……程序正确运行了，于是就保留了下来。几个星期以后，make拥有了几个用户，大部分是我的朋友，我又不愿破坏代码的基本结构。很遗憾，后来的事情你都知道了。”

百宝箱

make减轻了编译文件时的痛苦，但如果你觉得它还不够自动化，可以试一试这个叫**autoconf**的工具：

http://www.gnu.org/software/autoconf/

autoconf可以用来生成makefile。C程序员经常创造工具来自动化软件开发，这类工具在GNU网站上的数量越来越多。

---

① ant和rake分别是基于Java和Ruby的构建工具。——译者注

make冰箱贴

嘿，宝贝，如果你跟不上节奏，一定会爱上Head First酒吧的人写的这个程序！oggswing程序读取Ogg格式的音乐文件，然后创建一首摇摆乐。宝贝！看看你能否完成makefile，它先编译oggswing，然后用它来转换*.ogg*文件。

把whitennerdy.ogg转化为swing.ogg。

```
oggswing: ..........................................................

..........................................................

swing.ogg: ..........................................................

..........................................................
```

oggswing whitennerdy.ogg swing.ogg

gcc oggswing.c -o oggswing

whitennerdy.ogg

oggswing

oggswing.h

[SPACES]

oggswing.c

[TAB]

[SPACES]

[TAB]

make冰箱贴解答

嘿，宝贝，如果你跟不上节奏，一定会爱上Head First酒吧的人写的这个程序！oggswing程序读取Ogg格式的音乐文件，然后创建一首摇摆乐。宝贝！你将完成makefile，它先编译oggswing，然后用它来转换*.ogg*文件。

```
oggswing: oggswing.c oggswing.h
[TAB] gcc oggswing.c -o oggswing

swing.ogg: whitennerdy.ogg oggswing
[TAB] oggswing whitennerdy.ogg swing.ogg
```

[SPACES]

[SPACES]

百宝箱

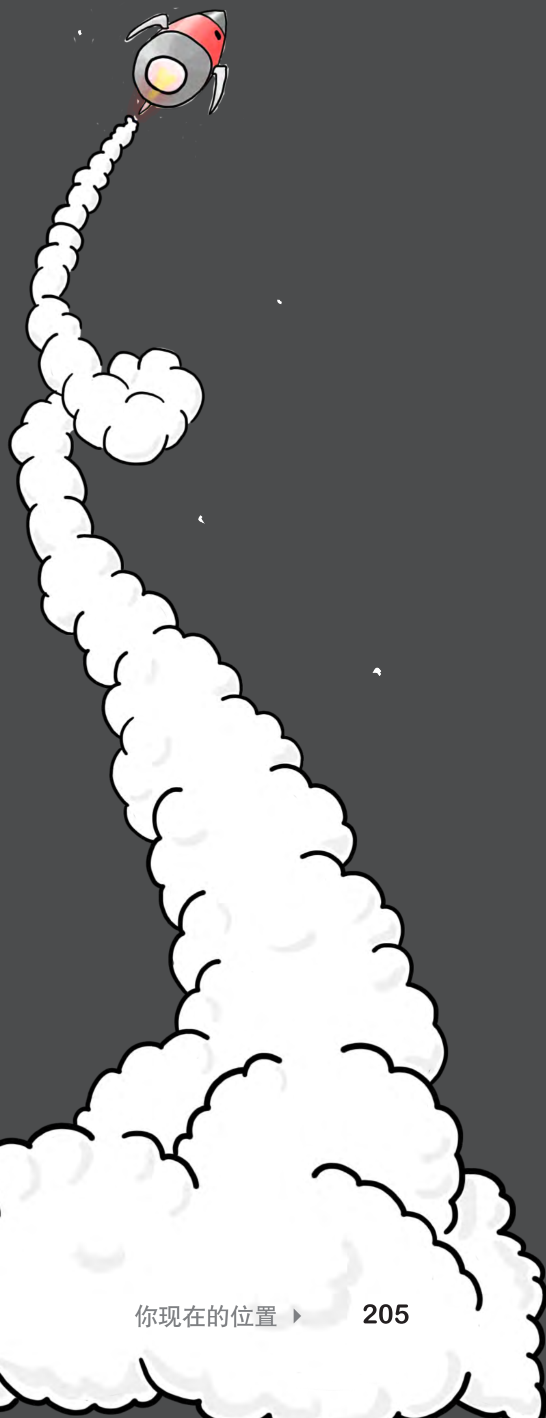
make可以做得更多，但是我们没有空间在这里讨论。关于make的更多信息和功能，请浏览 *GNU Make Manual*：

http://tinyurl.com/yczmjx

火箭升空！

如果你的代码构建起来很慢，使用make就可以提高速度。大多数开发人员习惯用make构建代码，就连小程序也不放过。有了make，就好比有一位一丝不苟的程序员坐在你身边，即使有再多的代码，make也会在需要的时候构建你需要的那些代码。

按时完成任务有时很重要……



要点

- 编译大量文件非常的耗时。
- 可以把目标代码保存在*.o文件中，加快编译速度。
- gcc不但能从源文件，而且能从目标文件编译程序。
- make工具可以用来自动化代码的构建过程。
- make清楚文件之间的依赖关系，可以只编译那些修改过的文件。
- 你需要使用makefile告诉make如何构建代码。
- 处理makefile的格式时要小心，别忘了用tab来缩进，不是空格！

C语言工具箱

学完第4章，现在你的工具箱中多出了数据类型和头文件。关于本书提示工具条的完整列表，请见附录ii。

char是数值。

小整数用short。

普通整数用int。

大整数用long。

一般的浮点数用float。

高精度的浮点数用double。

函数的声明与定义分离。

把声明放在头文件中。

用#include<>包含标准库头文件。

用#include “”包含本地头文件。

把目标代码保存到文件中，提高构建速度。

使用make管理代码构建过程。

实验人：　　　　实验日期：

C语言实验室1

Arduino

本实验会给你一份说明书，它描述了一个程序，你需要运用你在前几章中学到的知识构建这个程序。

这个项目比你之前见识到的项目都要大，所以动手之前请阅读完全部内容，并给自己一点时间。不要担心会被难倒，这里没有新概念，你也可以接着往后读，回过头再来做这个实验。

我们还为你补充了一些设计上的细节，万事俱备，你甚至可以搭建物理设备。

该去实现程序了，我们不会给你任何代码或答案。

说明书：让盆栽说话

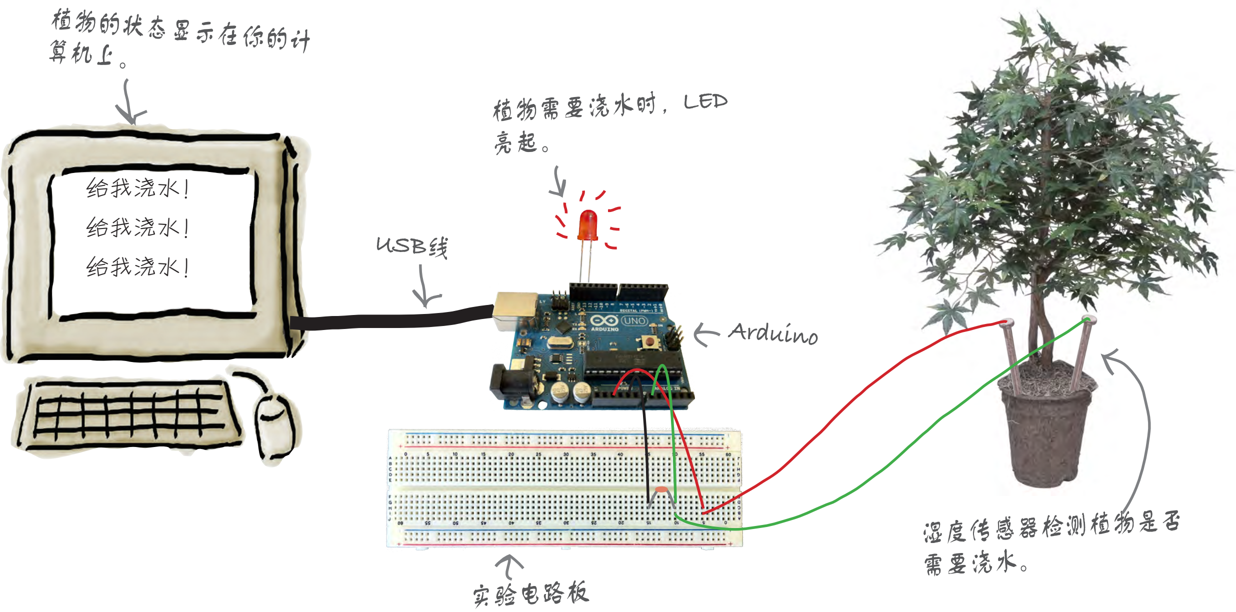
你可曾想过，你的植物告诉你它需要浇水？有了Arduino，植物就可以开口了！本实验中，你将创建一个由Arduino驱动的植物监控器，全用C语言来写。

需要构建以下这些东西。

物理设备

植物监控仪用湿度传感器测量植物土壤的湿度。当植物需要浇水时，LED就会亮起，并重复向你的计算机发送字符串“给我浇水”，直到你给植物浇水为止。

等你给植物浇了水，LED灯就会关闭，然后向计算机发送字符串“谢谢啊！”。



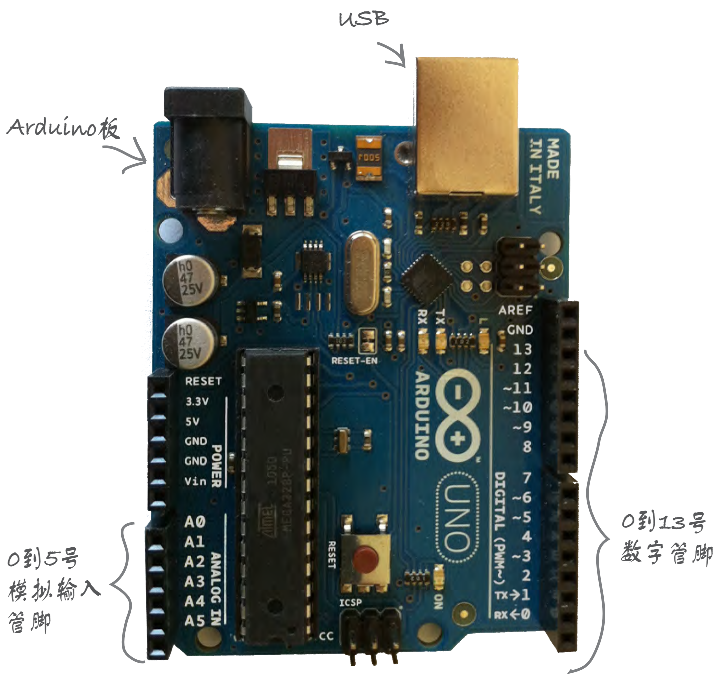
Arduino

植物监控仪的核心是Arduino。Arduino是一个基于微控制器的小型开源平台，它可以用来设计电子设备的原型。你可以在Arduino上面连接传感器，采集真实世界中的信息，Arduino的执行器会做出反应。整个过程需要用C代码来控制。

Arduino板上有14个数字IO管脚，它们用来输入和输出数据。我们可以用这些管脚来读取数据或控制执行器。

板上还有6个模拟输入管脚，可以从传感器读取电压值。

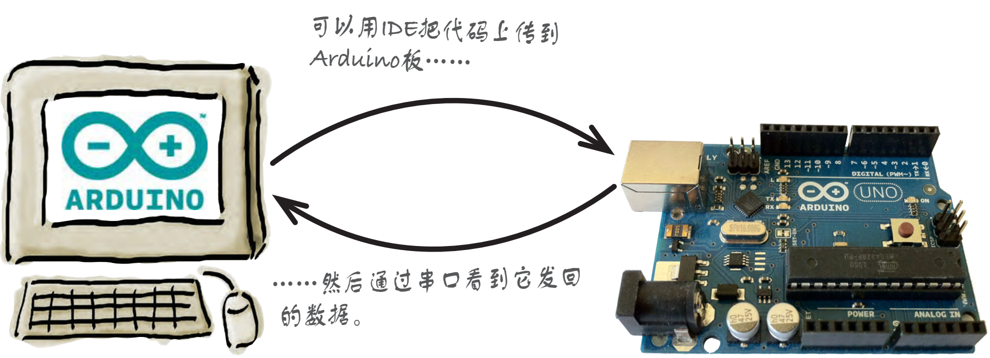
Arduino板用计算机的USB端口供电。



Arduino IDE

C代码在Arduino IDE中写，你可以用Arduino IDE检验代码是否正确并编译代码，然后通过计算机的USB端口把代码上传到Arduino。IDE还自带一个串口监视器，用来查看Arduino发回的数据（如果有的话）。

Arduino IDE是免费的，可以在*www.arduino.cc/en/Main/Software*下载。



装配物理设备

第一步是装配物理设备。虽然这一步是可选的，但我们强烈建议你试一试，你的植物会为此感激你的。

你需要：

- 1块Arduino
- 1块实验电路板
- 1个LED灯
- 1个10K欧姆电阻
- 2枚镀锌钉
- 3根短跨接线
- 2根长跨接线

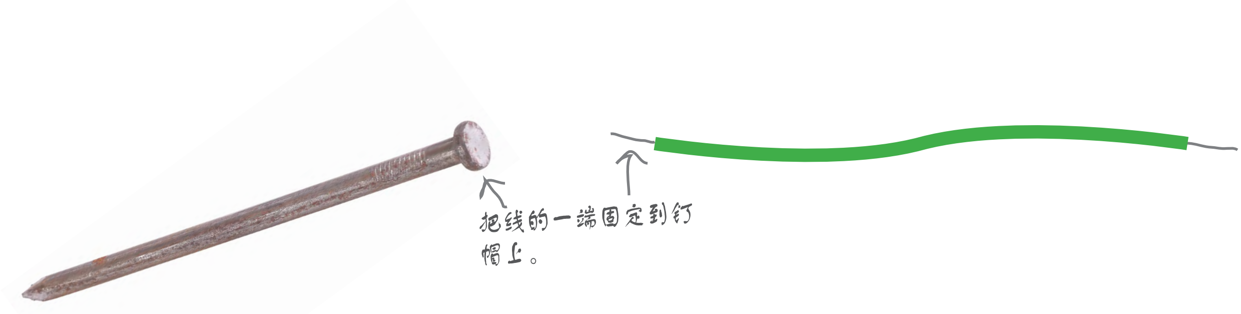
我们用Arduino Uno。

制作湿度传感器

取一根长跨接线，将其连接到一枚镀锌钉的钉帽上，可以绕在上面，也可以焊上去。

然后再取一根长跨接线，与另一枚镀锌钉相连。

湿度传感器会测量两枚镀锌钉之间的导电率，导电率高表示湿度高，导电率低说明湿度低。

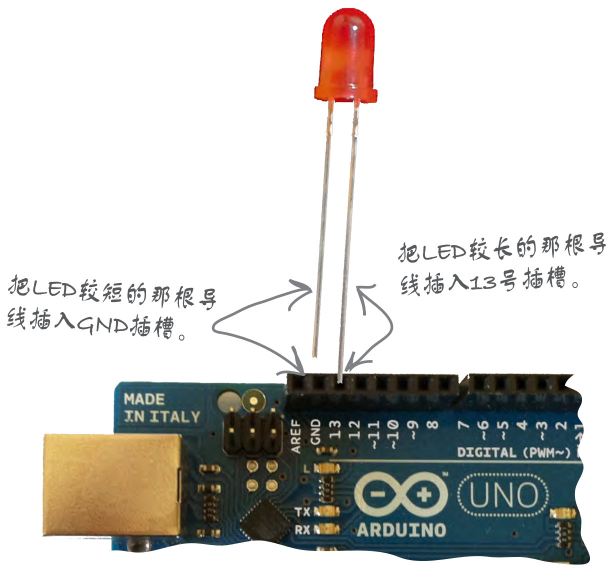


连接LED

仔细观察LED，你会发现它一根导线长（正极），一根导线短（负极）。

凑近Arduino，可以看到14道插槽沿着边缘一字排开，分别代表0~13号数字管脚，旁边还有一道插槽，标的是GND。把LED较长的那根导线（正极）插入13号插槽，较短的那根导线（负极）插入GND插槽。

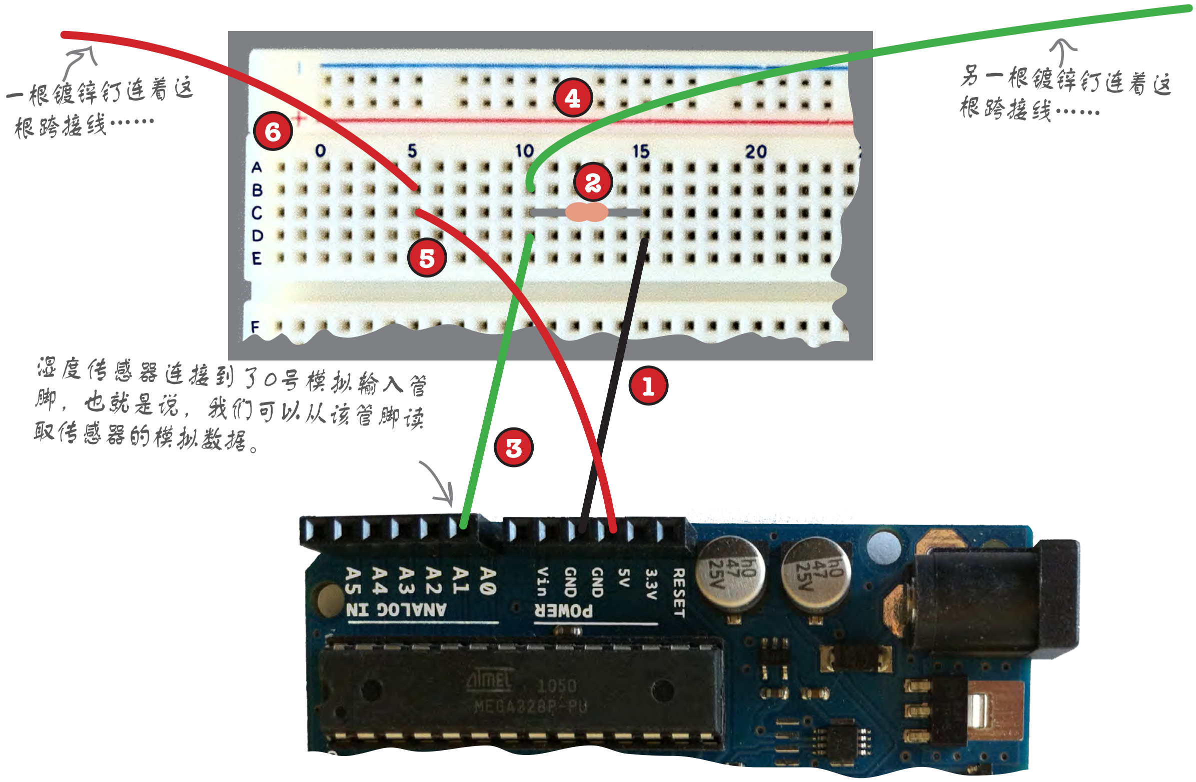
这样就可以通过13号管脚控制LED。



连接湿度传感器

湿度传感器的连接方法如下所示。

1. 用短跨接线将Arduino的GND管脚与实验电路板的D15插槽相连。
2. 10K欧姆电阻的一端与实验电路板C15插槽相连，另一端与C10插槽相连。
3. 用短跨接线将0号模拟输入管脚与实验电路板D10插槽相连。
4. 将镀锌钉上的跨接线与实验电路板B10插槽相连。
5. 用短跨接线将Arduino的5V管脚与实验电路板的C5插槽相连。
6. 将另一枚镀锌钉上的跨接线与实验电路板B5插槽相连。



现在，Arduino的物理装置已经组装好了，下面来看C代码……

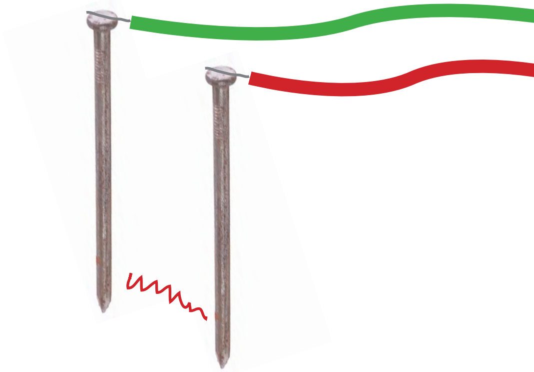
代码应该做

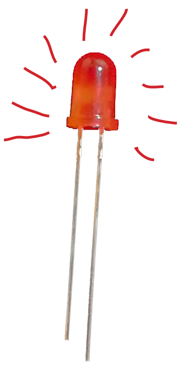
Arduino C代码应该做以下事情。

读取湿度传感器的数据

湿度传感器连到了模拟输入管脚，代码需要从该管脚读取模拟量。

在实验室，我们发现，一般当这个值低于800时，植物就需要浇水了。你种的植物可能不用，如果它是仙人掌的话。





把数据写到LED

LED连到了数字管脚。

当植物不需要浇水时，把数据写到LED连接的数字管脚，让它关闭LED。

当植物需要浇水时，把数据写到数字管脚，让它打开LED。如果你想做得更好，就让LED闪烁，还可以在数字接近800时让LED闪烁。

向串口写数据

当植物需要浇水时，需要重复地向计算机的串口写字符串“给我浇水！”。

当植物有了充足的水分，向串口写字符串“谢谢啊！”，写一次就行了。

假设Arduino已经插入了计算机的USB插口。

C代码怎么写

Arduino C程序有特定结构，必须这么写才行：

```
void setup()
{
/*程序启动时会调用这个函数，它对Arduino
板进行设置，把所有初始化代码都放在这里。*/
}
void loop()
{
/*这里写主代码。函数会不断循环，你可以用它
响应传感器的输入，直到Arduino板关闭。*/
}
```

除了这两个函数之外，还可以添加其他函数或声明，但是没有这两个函数，程序就不能工作。

用Arduino IDE来写Arduino C代码非常方便。你可以用它来检验代码是否正确并编译代码，然后把完整的程序上传到Arduino板，这样就可以看到运行结果了。

Arduino IDE还提供了一个Arduino函数库和一些有用的示例代码。我们在下一页列出了一些函数，对于写程序很有用。

几个有用的Arduino函数

在创建Arduino时需要用到下面这些函数。

`void pinMode(int pin, int mode)`

告诉Arduino数字管脚`pin`是输入还是输出，`mode`可以是`INPUT`或`OUTPUT`。

`int digitalRead(int pin)`

从数字管脚读取数据，返回值是`HIGH`或`LOW`。

`void digitalWrite(int pin, int value)`

把一个值写到数字管脚，`value`是`HIGH`或`LOW`。

`int analogRead(int pin)`

从模拟管脚读取一个值，返回值是0到1023的一个数。

`void analogWrite(int pin, int value)`

向某个管脚写模拟量，`value`是0到255的一个数。

`void Serial.begin(long speed)`

让Arduino以`speed`比特/秒的速率发送或接收串行数据，通常把`speed`设为9600。

`void Serial.println(val)`

向串口打印数据，`val`可以是任意数据类型。

`void delay(long interval)`

让程序暂停`interval`毫秒。

植物监控仪下线

Arduino项目的最后一步就是把湿度传感器插到植物的土壤中，并把Arduino连到计算机上，然后你就等着得到植物的最新状态吧。

如果你有Mac，并且想让你的植物开口说话，可以到Head First实验室网站下载一个脚本，它可以识别串行数据流，并大声地朗读出来：

www.headfirstlabs.com/books/hfc

5 结构、联合与位字段

创建自己的结构

生活可比数字复杂多了。

到目前为止，你只接触过C语言的基本数据类型，但如果想表示数字、文本以外的其他东西呢，或为现实世界中的事物建立模型，怎么办？结构将帮你创建自己的结构，模拟现实世界中错综复杂的事物。在本章中，你将学习如何把基本数据类型组成结构以及用联合处理生活的不确定性。如果你想简单地模拟"是"或"非"，可以用位字段。

有时要传很多数据

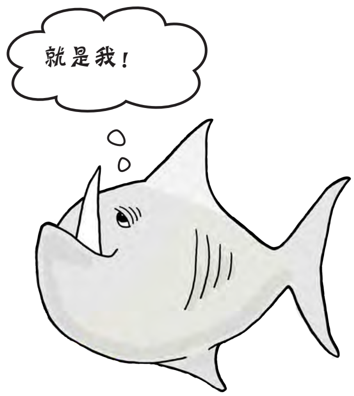
C语言可以处理很多不同类型的数据：小数字、大数字、浮点数、字符与文本。但现实世界中的事物往往需要一条以上的数据来记录。比如下面这个例子，两个函数处理同一个东西，因此需要接收相同的数据：

"const char *"表示将传递字符串字面值。

这两个函数接收相同的参数。

```
/* 打印目录项 */
void catalog(const char *name, const char *species, int teeth, int age)
{
  printf("%s is a %s with %i teeth. He is %i\n",
    name, species, teeth, age);
}

/* 打印贴在水缸上的标签 */
void label(const char *name, const char *species, int teeth, int age)
{
  printf("Name:%s\nSpecies:%s\n%i years old, %i teeth\n",
    name, species, teeth, age);
}
```

不算太坏，是吧？虽然只传了4条数据，但代码已经有点乱了：



两次都传了相同的4条数据。

```
int main()
{
  catalog("Snappy", "Piranha", 69, 4);
  label("Snappy", "Piranha", 69, 4);
  return 0;
}
```

只有一条鱼，但却传了4条数据。

怎么才能解决这个问题？只是想描述一样东西而已，有没有办法可以不用传那么多数据？

窃窃私语

Joe: 没错，虽然现在只有4条数据，但如果我们修改程序，给鱼多加一条数据呢？

Frank: 那也只多了1个参数而已。

Jill: 虽然只有一条数据，但我们要把它加到每一个接收鱼作参数的函数中。

Joe: 没错，对一个大程序来说，如果我们多加一条数据，这样的函数可能有上百个。

Frank: 说得好，那我们该怎么办呢？

Joe: 简单，只要把这些数据组合成一样东西就行了，类似数组。

Jill: 不知道这样行不行，数组通常保存相同类型的数据。

Joe: 没错。

Frank: 我懂了，我们现在要同时保存字符串和整型。所以我们不能把它们放进一个数组。

Jill: 我也觉得不能。

Joe: 但C语言一定有解决的方法，想想我们需要什么。

Frank: 嗯，我们需要的这样东西能让我们同时引用一组不同类型的数据，仿佛它们是一条数据。

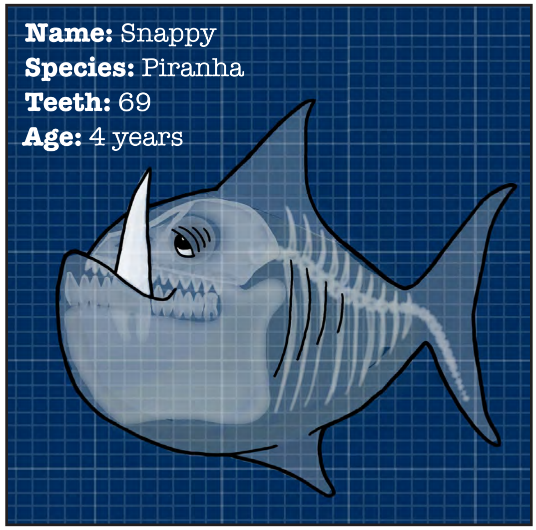
Jill: 我们应该还没见过这样的东西，对吗？

你需要一样东西，能让你在一条大数据中记录多条数据。

用结构创建结构化数据类型

如果需要把一批数据打包成一样东西，就可以使用结构（struct）。struct是structured data type（结构化数据类型）的缩写。有了结构，就可以像下面这样把不同类型的数据写在一起，封装成一个新的大数据类型：

```
struct fish {
    const char *name;
    const char *species;
    int teeth;
    int age;
};
```



这段代码会创建一个新的自定义数据类型，它由一批其他数据组成。事实上，结构与数组有些相似，除了以下两点：

- **结构的大小固定。**
- **结构中的数据都有名字。**

定义新结构以后，如何用它来创建数据？和新建数组很像，你只需要保证每条数据按照它们在结构中定义的顺序出现即可：

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
```

struct fish是数据类型。
snappy是变量名。
名字。
品种。
牙齿数。
snappy的年龄。

这里没有蠢问题

问： 喂，等等，什么是const char *来着？

答： const char *用来保存你不想修改的字符串，也就是字符串字面值。

问： fish结构会保存字符串吗？

答： 在这个例子中不会，这里的fish结构中只保存了字符串指针，也就是字符串的地址，字符串保存在存储器中其他位置。

问： 但还是可以把整个字符串保存在结构中吧？

答： 对，只要把字符串定义成字符数组就行了，像char name[20];。

只要把“鱼”给函数就行了

现在，你只要把新的自定义数据传给函数就行了，而不必传递一大批零散的数据。

```
/* 打印目录项 */
void catalog(struct fish f)
{
  ...
}

/* 打印贴在水缸上的标签 */
void label(struct fish f)
{
  ...
}
```

是不是简单多了？现在函数只需接收一条数据，而且调用函数的代码也更易读了：

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
catalog(snappy);
label(snappy);
```

以上便是定义自定义数据类型的方法，但怎么才能使用它们呢？函数如何读取结构中保存的某条数据呢？

把参数封装在结构中，代码会更稳定。

鱼的好处

把数据放在结构中传递有一个好处，就是修改结构的内容时，不必修改使用它的函数。比如要在fish中多加一个字段：

```
struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
  int favorite_music;
};
```

catalog()和label()知道有人会给它们一条fish，但却不知道fish中现在有了更多的数据，它们也不关心，只要fish有它们需要的所有字段就行了。

这就意味着，使用结构，不但代码更好读，而且能够更好地应对变化。

使用“.”运算符读取结构字段

因为结构和数组有些像，你可能以为能像读取数组元素那样读取结构字段：

```
struct fish snappy = {"Snappy", "piranha", 69, 4};
printf("Name = %s\n", snappy[0]);
```

既然snappy是数组指针，就可以像这样访问它的第一个字段。

如果像访问数组元素那样读取结构字段，会得到编译错误。

```
File Edit Window Help Fish
> gcc fish.c -o fish
fish.c: In function 'main':
fish.c:12: error: subscripted value is neither array nor pointer
>
```

但不可以这样做。尽管结构可以像数组那样在结构中保存字段，但读取时只能**按名**访问。可以使用“.”运算符访问结构字段。如果你用过JavaScript或Ruby这样的语言，一定会觉得非常眼熟：

```
struct fish snappy = {"Snappy", "piranha", 69, 4};
printf("Name = %s\n", snappy.name);
```

这是snappy中的name属性。

将返回字符串“Snappy”。

```
File Edit Window Help Fish
> gcc fish.c -o fish
> ./fish
Name = Snappy
>
```

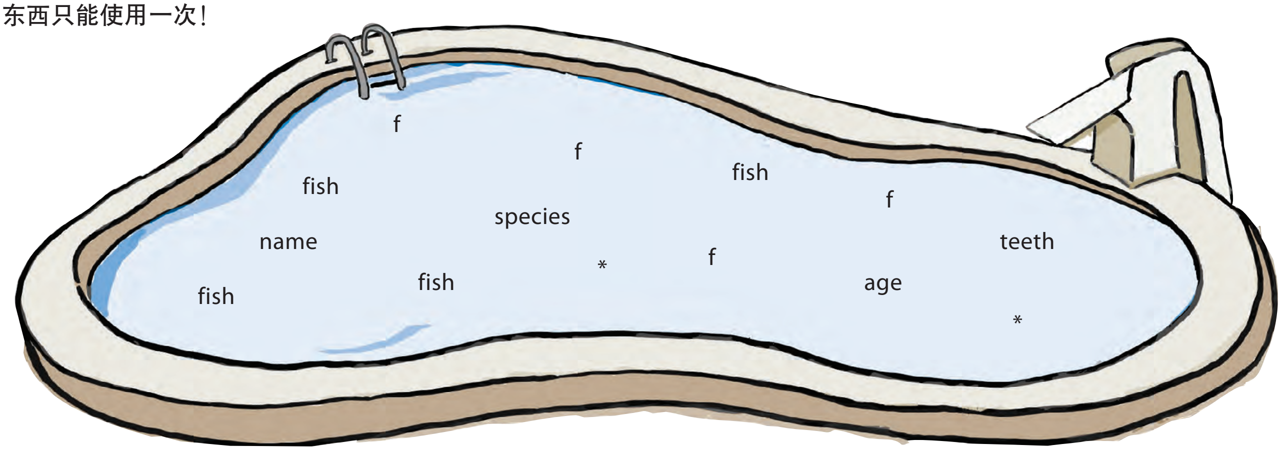
行了，既然你已经学会使用结构了，看看能否修改刚才的代码……

食人鱼~~游泳池~~拼图

你的任务是写一个新版的catalog()函数，函数将使用fish结构。从游泳池中取出代码片段，填入空白横线处。每个片段只能使用一次，有的可能一次都用不到。

```
void catalog(struct fish f)
{
  printf("%s is a %s with %i teeth. He is %i\n",
    ......·.........., ......·.........., ......·.........., ......·..........);
}

int main()
{
  struct fish snappy = {"Snappy", "Piranha", 69, 4};
  catalog(snappy);
  /* 暂时先跳过调用label函数的代码 */
  return 0;
}
```

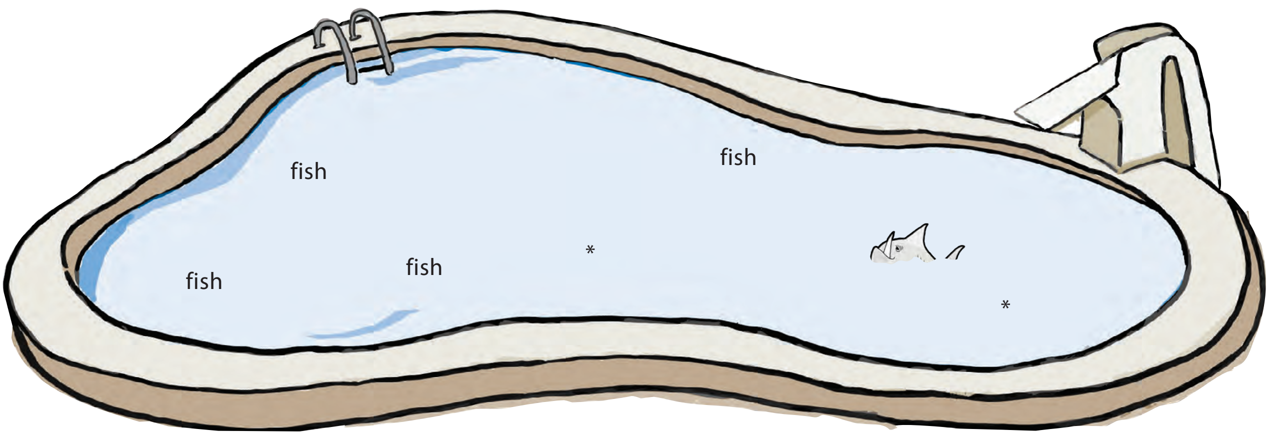
注意：游泳池中的每样东西只能使用一次！

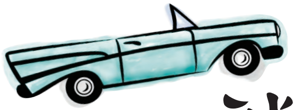


食人鱼 ~~游泳池~~拼图解答

你的任务是写一个新版的catalog()函数，函数将使用fish结构。请从游泳池中取出代码片段，填入空白横线处。

```
void catalog(struct fish f)
{
  printf("%s is a %s with %i teeth. He is %i\n",
     f.name, f.species, f.teeth, f.age);
}

int main()
{
  struct fish snappy = {"Snappy", "Piranha", 69, 4};
  catalog(snappy);
  /* 暂时先跳过调用label函数的代码  */
  return 0;
}
```





试驾

你已经重写了catalog()函数，重写label()对你来说也是小菜一碟。写完以后就可以编译代码，检查它能否正确运行：

快看，有人在用make……

catalog()函数打印了这行。

label()函数打印了这几行。

```
File Edit Window Help FishAreFriendsNotFood
> make pool_puzzle && ./pool_puzzle
gcc pool_puzzle.c -o pool_puzzle
Snappy is a Piranha with 69 teeth. He is 4
Name:Snappy
Species:Piranha
4 years old, 69 teeth
>
```

好极了！代码和刚才一样能够正确运行，不同的是，这次调用函数的代码变得异常简洁：

```
catalog(snappy);

label(snappy);
```

代码的可读性提高了，而且当你决定在结构中保存额外的数据时，不必修改使用结构的函数。

这里没有蠢问题

问： 结构就是数组吗？

答： 不是数组，不过结构把多条数据组合在一起，这点和数组很像。

问： 数组变量就是一个指向数组的指针，那么结构变量是一个指向结构的指针吗？

答： 不是，结构变量是结构本身的名字。

问： 我可以用下标[0]、[1]……访问结构字段吗？

答： 不可以，你只能按名访问。

问： 结构就相当于其他语言中的类？

答： 它们很相似，但在结构中添加方法可就没那么容易了。

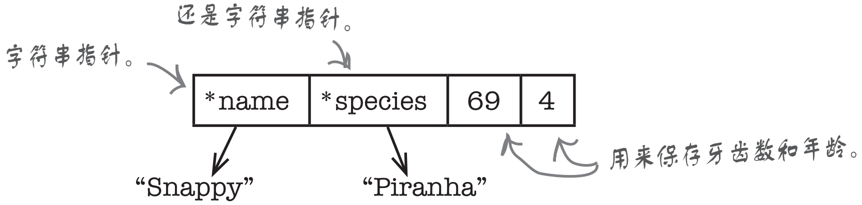
聚焦存储器中的结构

在定义结构时，你并没有让计算机在存储器中创建任何东西，只是给了计算机一个模板，告诉它你希望新的数据类型长什么样子。

```
struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
};
```

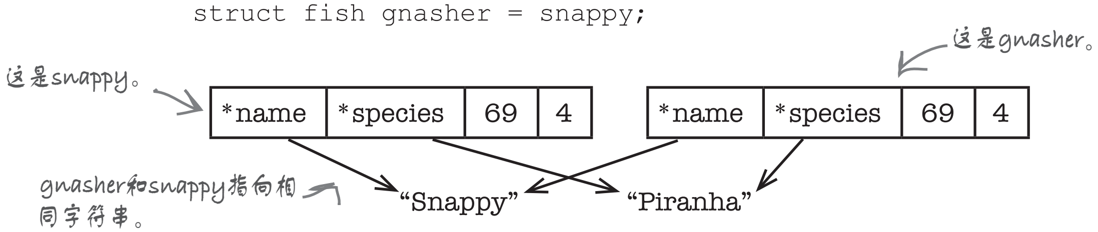
当定义新变量时，计算机则需要在存储器中为结构的实例创建空间，这块空间必须足够大，以装下结构中的所有字段：

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
```



那么当把一个结构变量赋给另一个结构变量时会发生什么？计算机会创建一个**全新的结构副本**，也就是说，计算机需要再分配一块存储器空间，大小和原来相同，然后把每个字段都复制过去。

```
struct fish snappy = {"Snappy", "Piranha", 69, 4};
struct fish gnasher = snappy;
```



切记，为结构变量赋值相当于叫计算机复制数据。

小心！

复制的是指向字符串的指针，而非字符串本身。

当把一个结构变量赋给另一个结构变量，计算机会复制结构的内容。如果结构中含有指针，那么复制的仅仅是指针的值，像这里，gnasher和snappy的name和species字段指向相同字符串。

结构中的结构

别忘了，定义结构其实就是在创造新的数据类型。C语言有很多内置数据类型，例如int和short，但有了结构，我们就可以把现有类型组合起来，向计算机描述更复杂的东西。

既然结构可以用现有数据类型创建数据类型，也就能用其他结构创建结构。具体怎么做？来看一个例子。

```
struct preferences {      <- 鱼喜欢这些东西。
  const char *food;
  float exercise_hours;
};

struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
  struct preferences care;
};
```

（注：“struct preferences care;”旁的手写批注——“新字段。”；“结构中的结构。”；“这叫嵌套（nesting）。”；“我们的新字段叫care，它包含由preferences结构定义的字段。”）

> **为什么要嵌套定义结构？**
>
> 之所以要这么做是为了对抗**复杂性**。通过使用结构，我们可以建立更大的数据块。通过把结构组合在一起，我们可以创建更大的数据结构。本来你只能用int、short，但有了结构以后，就可以描述十分复杂的东西，比如**网络流**和**视频图像**。

上述代码向计算机描述了一个结构中的结构。你可以像之前一样用数组语法创建变量，但现在可以在数据中包含结构中的结构。

```
struct fish snappy = {"Snappy", "Piranha", 69, 4, {"Meat", 7.5}};
```

（注：手写批注——{"Meat", 7.5}：“保存在care字段中的的结构数据。”；"Meat"：“care.food的值。”；7.5：“care.exercise_hours的值。”）

一旦把结构组合起来，就可以使用一连串的“.”运算符来访问字段：

```
printf("Snappy 喜欢吃 %s", snappy.care.food);
printf("Snappy 喜欢锻炼 %f hours", snappy.care.exercise_hours);
```

快试试你的新技能吧……

Head First水族馆的人开始登记鱼的信息，每条鱼都有很多数据要记录，这是它们的结构：

```
struct exercise {
  const char *description;
  float duration;
};

struct meal {
  const char *ingredients;
  float weight;
};

struct preferences {
  struct meal food;
  struct exercise exercise;
};

struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
  struct preferences care;
};
```

其中某条鱼有以下数据要记录：

```
Name: Snappy
Species: Piranha
Food ingredients: meat
Food weight: 0.2 lbs
Exercise description: swim in the jacuzzi
Exercise duration 7.5 hours
```

问题0：这条数据用C语言怎么表示？

```
struct fish snappy = ..........................................................
```

问题1：补全`label()`函数的代码，输出以下信息：

```
Name:Snappy
Species:Piranha
4 years old, 69 teeth
Feed with 0.20 lbs of meat and allow to swim in the jacuzzi for 7.50 hours
```

```
void label(struct fish a)
{
  printf("Name:%s\nSpecies:%s\n%i years old, %i teeth\n",
         a.name, a.species, a.teeth, a.age);
  printf("Feed with %2.2f lbs of %s and allow to %s for %2.2f hours\n",
         .................................... , .................................... ,
         .................................... , .................................... );
}
```

练习解答

Head First水族馆的人开始登记鱼的信息，每条鱼都有很多数据要记录，这是它们的结构：

```
struct exercise {
  const char *description;
  float duration;
};

struct meal {
  const char *ingredients;
  float weight;
};

struct preferences {
  struct meal food;
  struct exercise exercise;
};

struct fish {
  const char *name;
  const char *species;
  int teeth;
  int age;
  struct preferences care;
};
```

其中某条鱼有以下数据要记录：

```
Name: Snappy
Species: Piranha
Food ingredients: meat
Food weight: 0.2 lbs
Exercise description: swim in the jacuzzi
Exercise duration 7.5 hours
```

问题0：这条数据用C语言怎么表示？

```
struct fish snappy = { "Snappy, "Piranha" ,69,4,{{ "meat" ,0.2},{ "swim in the jacuzzi" ,7.5}}};
```

问题1：补全label()函数的代码，输出以下信息：

```
Name:Snappy
Species:Piranha
4 years old, 69 teeth
Feed with 0.20 lbs of meat and allow to swim in the jacuzzi for 7.50 hours
```

```
void label(struct fish a)
{
  printf("Name:%s\nSpecies:%s\n%i years old, %i teeth\n",
         a.name, a.species, a.teeth, a.age);
  printf("Feed with %2.2f lbs of %s and allow to %s for %2.2f hours\n",
         a.care.food.weight , a.care.food.ingredients ,
         a.care.exercise.description , a.care.exercise.duration );
}
```

用typedef为结构命名。

当创建内置数据类型变量时，只要写int或double就行了，但每次创建结构变量时，不得不加上struct关键字。

```
struct cell_phone {
  int cell_no;
  const char *wallpaper;
  float minutes_of_charge;
};
...
struct cell_phone p = {5557879, "sinatra.png", 1.35};
```

在C语言中可以为结构创建**别名**，你只要在struct关键字前加上**typedef**，并在右花括号后写上**类型名**，就可以在任何地方使用这种新类型。

typedef表示将为结构类型起一个新的名字。

```
typedef struct cell_phone {
  int cell_no;
  const char *wallpaper;
  float minutes_of_charge;
} phone;
...
phone p = {5557879, "sinatra.png", 1.35};
```

phone将成为"struct cell_phone"的别名。

现在，只要编译器看到"phone"，就把它当成"struct cell_phone"。

typedef可以用来缩短代码长度，并让代码更容易阅读。试试在代码中加入typedef……

新类型叫什么？

当你用typedef为结构创建别名，需要决定别名叫什么。别名其实就是类型名，也就是说结构有两个名字：一个是结构名（struct **cell_phone**），另一个是**类型**名（**phone**）。为什么要有两个名字？一般一个就够了。如果只写类型名而不写结构名，编译器也没意见：

```
typedef struct {
  int cell_no;
  const char *wallpaper;
  float minutes_of_charge;
} phone;
phone p = {5557879, "s.png", 1.35};
```

这是别名。

练习

潜水员要开始巡逻水池，他需要给潜水服贴上新的标签。问题是一些代码不见了，你能写出来吗？

```
#include <stdio.h>

 ........................struct {
  float tank_capacity;
  int tank_psi;
  const char *suit_material;
} ........................;

 ........................struct scuba {
  const char *name;
  equipment kit;
} diver;

void badge(........................ d)
{
  printf("Name: %s Tank: %2.2f(%i) Suit: %s\n",
    d.name, d.kit.tank_capacity, d.kit.tank_psi, d.kit.suit_material);
}

int main()
{
  ........................ randy = {"Randy", {5.5, 3500, "Neoprene"}};
  badge(randy);
  return 0;
}
```

练习解答

潜水员要开始巡逻水池，他需要给潜水服贴上新的标签。问题是一些代码不见了，你写出来了吗？

```
#include <stdio.h>

  typedef struct {
  float tank_capacity;
  int tank_psi;
  const char *suit_material;
} equipment;

  typedef struct scuba {
  const char *name;
  equipment kit;
} diver;

void badge(diver d)
{
  printf("Name: %s Tank: %2.2f(%i) Suit: %s\n",
    d.name, d.kit.tank_capacity, d.kit.tank_psi, d.kit.suit_material);
}

int main()
{
  diver randy = {"Randy", {5.5, 3500, "Neoprene"}};
  badge(randy);
  return 0;
}
```

程序员给结构取了个名字叫scuba，但你只会用类型名diver。

要点

- 结构是一种由一系列其他数据类型组成的数据类型。
- 结构的大小固定。
- 结构字段按名访问，用<结构>.<字段名>语法（也叫“点表示法”）。
- 结构字段在存储器中保存的顺序和它们出现在代码中的顺序相同。
- 可以嵌套定义结构。
- `typedef`创建数据类型的别名。
- 用`typedef`定义结构时可以省略结构名。

这里没有蠢问题

问： 结构字段在存储器中是紧挨着摆放的吗？

答： 有时两个字段之间会有小的空隙。

问： 为什么？

答： 计算机总是希望数据能对齐字边界（word boundary）。如果计算机的字长是32位，就不希望某个变量（比如short）跨越32位的边界保存。

问： 所以计算机会留下一道空隙，然后在下一个32位字开始的地方保存`short`？

答： 是的。

问： 也就是说，每个字段都占用一整个字？

答： 不一定，计算机在两个字段之间留出空隙仅仅是为了防止某个字段跨越字边界。如果几个字段能放在一个字中，计算机就会那么做。

问： 为什么计算机如此在意字边界？

答： 计算机按字从存储器中读取数据，如果某个字段跨越了多个字，CPU就必须读取多个存储器单元，并以某种方式把读到的值合并起来。

问： 这样会很慢吗？

答： 会很慢。

问： 在Java那样的语言中，如果我把对象赋给变量，它不会复制对象，仅仅复制引用，为什么C语言不这样做？

答： 在C语言中，所有赋值都会复制数据，如果你想复制数据的引用，就应该赋指针。

问： 结构名的问题我还没搞清楚，什么是结构名？什么是别名？

答： 结构名是`struct`关键字后面的那个单词。假设你写的是`struct peter_parker{ ... }`，那么结构名就是`peter_parker`，当创建变量时，你会写`struct peter_ parker x`。

问： 那别名呢？

答： 有时，你不想在声明变量时还使用`struct`关键字，那么就可以用`typedef`创建别名。`typedef struct peter_parker { ... } spider_man;`里面的`spider_man`就是别名。

问： 什么是匿名结构？

答： 匿名结构就是没有名字的结构，`typedef struct { ... } spider_man;`有一个叫`spider_man`的别名，但没有结构名。很多时候，如果创建了别名，也就不需要结构名了。

如何更新结构

结构其实就是把一组绑在一起的变量当成一条数据处理。你已经学会了创建结构对象，并使用“点表示法”访问结构的值，那么怎样修改结构中已经存在的某个值呢？可以像修改变量那样修改字段：

创建了一个结构。

```
fish snappy = {"Snappy", "piranha", 69, 4};
printf("Hello %s\n", snappy.name);
snappy.teeth = 68;
```

设置teeth字段的值。

读取name字段的值。

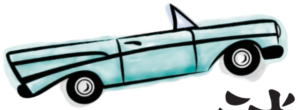
呀！看来Snappy咬到了硬东西。

既然如此，你应该能分析出下面这段代码做了什么。

```
#include <stdio.h>

typedef struct {
  const char *name;
  const char *species;
  int age;
} turtle;

void happy_birthday(turtle t)
{
  t.age = t.age + 1;
  printf("Happy Birthday %s! You are now %i years old!\n",
    t.name, t.age);
}

int main()
{
  turtle myrtle = {"Myrtle", "Leatherback sea turtle", 99};
  happy_birthday(myrtle);
  printf("%s's age is now %i\n", myrtle.name, myrtle.age);
  return 0;
}
```

乌龟Myrtle

但奇怪的事情发生了……



试驾

你编译并运行了代码，结果如下。

TMD，咋回事？

甜蜜的

```
File Edit Window Help ILikeTurtles
> gcc turtle.c -o turtle && ./turtle
Happy Birthday Myrtle! You are now 100 years old!
Myrtle's age is now 99
>
```

奇怪的事情发生了。

这段代码创建了一个新的结构，然后把它传给了一个函数，按理说，函数会将结构中某个字段的值递增1，但程序却……。

你知道age字段在happy_birthday()函数中更新了，因为printf()函数显示了递增以后age的值，但奇怪的是，虽然happy_birthday()更新了age，但程序返回main()函数以后，age又变回了原来的值。

脑力风暴

代码的行为十分诡异，但你已经掌握了足够多的信息，应该能推测到底发生了什么，是吧？

代码克隆了乌龟

仔细看这段代码，它调用了happy_birthday()函数：

当把一个结构赋给另一个时，结构的值复制到了新结构中。

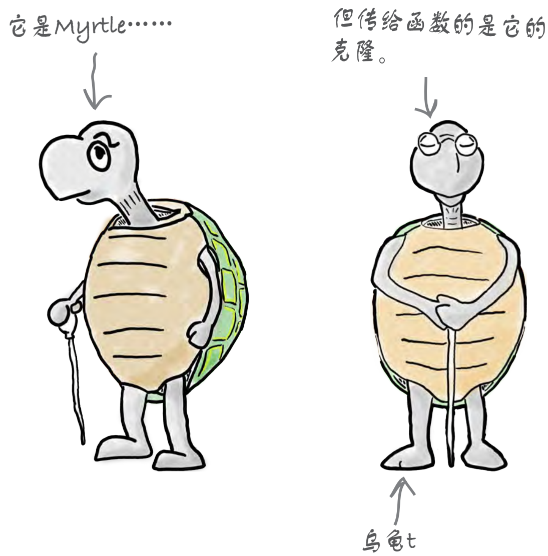
```
void happy_birthday(turtle t)
{
  ...
}

...
happy_birthday(myrtle);
```

这是我们传给函数的那只乌龟。

myrtle结构会复制给形参。

在C语言中，参数按值传递给函数。也就是说，当调用函数时，传入函数的值会赋给形参，因此这段代码等价于：

```
turtle t = myrtle;
```



别忘了，在C语言中，当为结构赋值时，计算机会复制结构的值。当调用happy_birthday()函数时，形参t中放的是myrtle结构的副本，仿佛函数克隆了原来那只乌龟，于是函数中的代码虽然更新了乌龟的年龄，却不是原来的那只。

函数返回以后呢？形参t不见了，main()中剩下的代码使用了myrtle结构。而myrtle的值从来没有被代码修改过，它一直是一条完全独立的数据。

如果想把结构传给函数并在函数中更新它的值，该怎么做？

你需要结构指针

当把变量传给scanf()函数时，不能把变量本身传给scanf()，而是要传一个指针：

```
scanf("%f", &length_of_run);
```

为什么要用指针？因为只有把变量在存储器中的位置告诉函数，函数才能更新保存在那里的数据，才能更新变量。

你也可以这样更新结构。如果想让函数更新结构变量，就不能把结构作为参数传递，因为这样做仅仅是将数据的副本复制给了函数。取而代之，你可以传递结构的地址：

```
void happy_birthday(turtle *t)
{
  ...
}

...
happy_birthday(&myrtle);
```

表示“有人要给我一个结构指针”。

别忘了，指针即地址。

你将把myrtle变量的地址传给函数。

磨笔上阵

你能补出新版happy_birthday()函数中的表达式吗？

注意，t现在是指针变量。

```
void happy_birthday(turtle *t)
{
  ............age = ............age + 1;
  printf("Happy Birthday %s! You are now %i years old!\n",
        ............name, ............age);
}
```

磨笔上阵 解答

请补全新版happy_birthday()函数中的的表达式。

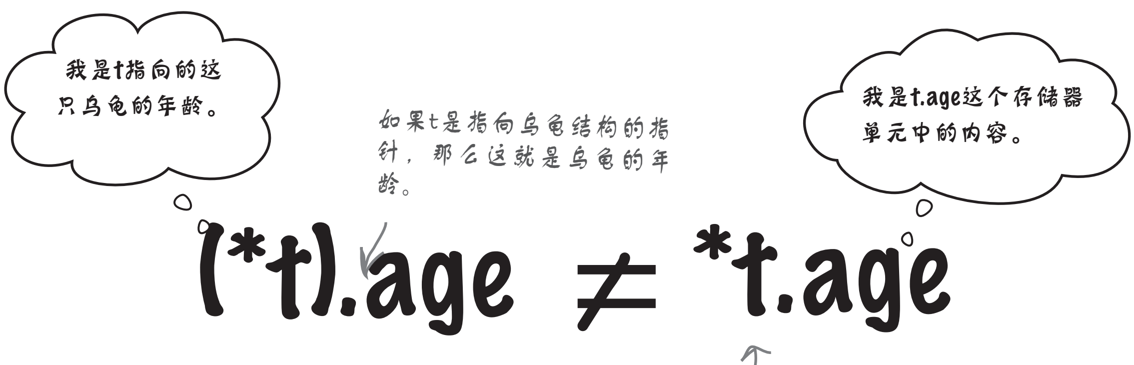
```
void happy_birthday(turtle *t)
{
   (*t).age = (*t).age + 1;
   printf("Happy Birthday %s! You are now %i years old!\n",
         (*t).name, (*t).age);
}
```

要把*放在变量名前，因为你想得到指针指向的值。

括号非常重要，如果不加会出错。

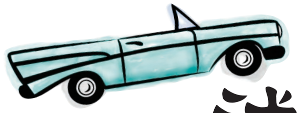
(*t).age和*t.age

为什么*t外面一定要加括号？因为(*t).age与*t.age完全是两个不同的表达式。



表达式***t.age**等于***(t.age)**。请思考一下表达式***(t.age)**的含义。它代表“t.age这个存储器单元中的内容”，但t.age不是存储器单元。

使用结构时要小心括号的位置，它们会影响表达式的值。



试驾

检查程序有没有错误：

```
File Edit Window Help ILikeTurtles
> gcc happy_birthday_turtle_works.c -o happy_birthday_turtle_works
> ./happy_birthday_turtle_works
Happy Birthday Myrtle! You are now 100 years old!
Myrtle's age is now 100
>
```

太棒了，现在函数正确工作了。
通过传递结构指针，函数更新了原来的数据，而不是修改本地的副本。

t->age代表(*t).age。

我知道新代码是怎么工作的，但括号和*表示法这些东西使代码不那么好读，有没有什么解决方法？

是的，还有一种表示结构指针的方法，它更易于阅读。
为了把括号放对地方，在处理指针时需要非常谨慎，因此C语言的发明者设计了一种更简洁、更易于阅读的语法。下面两个表达式含义相同：

```
(*t).age

t->age
```

这两个表达式含义相同。

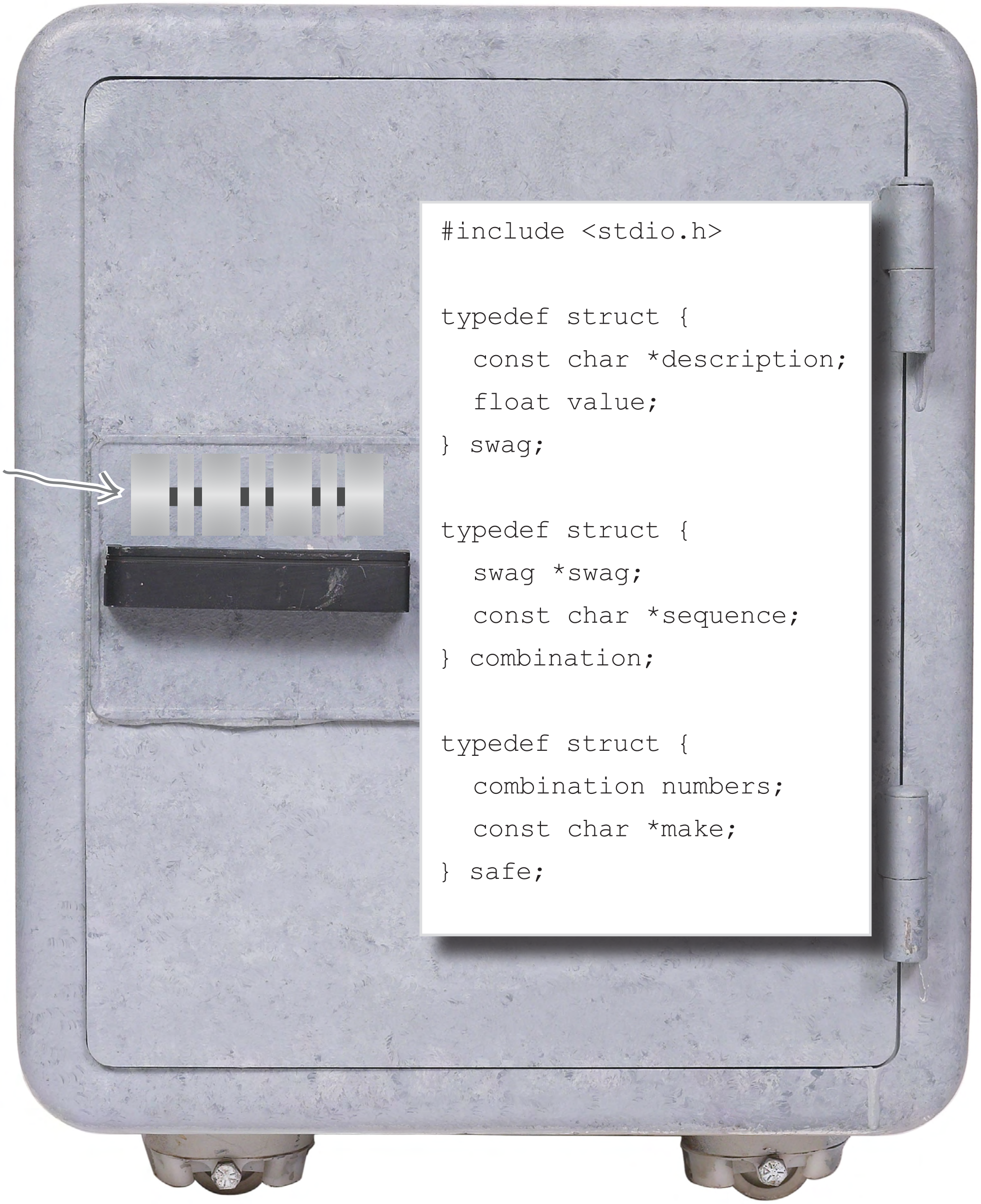
t->age表示“由t指向的结构中的age字段”，也就是说happy_birthday()函数还能这么写：

```
void happy_birthday(turtle *a)
{
  a->age = a->age + 1;
  printf("Happy Birthday %s! You are now %i years old!\n",
         a->name, a->age);
}
```

保险箱窃贼

嘘……夜深了，这里是银行的金库。你能把密码轮旋转到正确的位置，破解保险箱吗？研究以下代码，看能不能找到正确的组合，偷到金子。小心！有个类型叫swag，有个字段也叫swag。

需要破解这个组合。



```
#include <stdio.h>

typedef struct {
  const char *description;
  float value;
} swag;

typedef struct {
  swag *swag;
  const char *sequence;
} combination;

typedef struct {
  combination numbers;
  const char *make;
} safe;
```

银行用以下语句创建了保险箱：

```
swag gold = {"GOLD!", 1000000.0};
combination numbers = {&gold, "6502"};
safe s = {numbers, "RAMACON250"};
```

哪种组合能让你得到字符串“GOLD!”？从每一栏中选择一个单词或符号，组成表达式。

con	.	s	+	swag	.	value
s	->	numbers	.	description	-	swag
numbers	:	swag	->	value	->	description
swap	-	gold	-	sequence	+	gold

这里没有蠢问题

问： 为什么计算机要把值复制给形参变量？

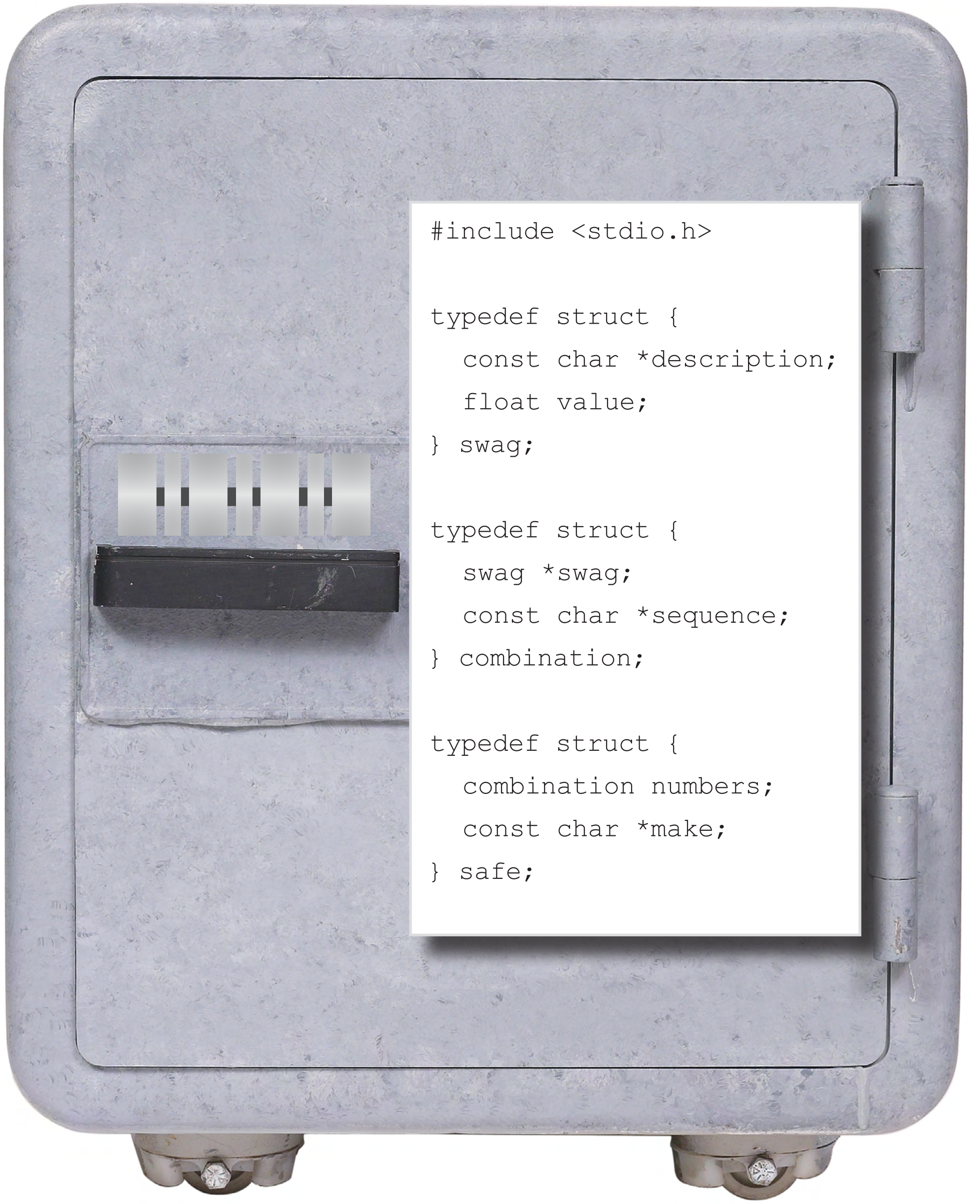
答： 计算机通过把值赋给函数形参的方式向函数传值，所有赋值都会复制值。

问： 为什么`*t.age`与`(*t).age`的含义不同？

答： 因为计算机先对“点”运算符求值，然后对*运算符求值。

保险箱窃贼解答

嘘……夜深了，这里是银行的金库。你把密码轮旋转到正确的位置，破解了保险箱。你研究了以下代码，找到了正确的组合，顺利偷到了金子。



```c
#include <stdio.h>

typedef struct {
  const char *description;
  float value;
} swag;

typedef struct {
  swag *swag;
  const char *sequence;
} combination;

typedef struct {
  combination numbers;
  const char *make;
} safe;
```

银行用以下语句创建了保险箱：

```
swag gold = {"GOLD!", 1000000.0};
combination numbers = {&gold, "6502"};
safe s = {numbers, "RAMACON250"};
```

哪种组合能让你得到字符串“GOLD!”？从每一栏中选择一个单词或符号，组成表达式。

con	.	s	+	swag	.	value
s	->	numbers	.	description	-	swag
numbers	:	swag	->	value	->	description
swap	-	gold	-	sequence	+	gold

可以用以下命令显示保险箱中的金子：

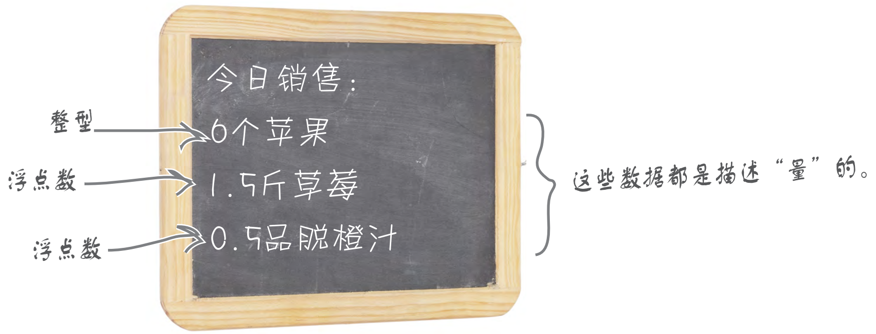
printf("Contents = %s\n", s.numbers.swag->description);

要点

- 当调用函数时，计算机会把值**复制**给形参变量。
- 可以像创建其他类型的指针那样创建结构指针。
- “指针->字段”等于“(*指针).字段”。
- “->”表示法省掉了括号，代码更易阅读。

同一类事物，不同数据类型

可以用结构来模拟现实世界中错综复杂的事物，但有些数据不止一种数据类型：



假如想记录某样东西的“量”，既可以用**个数**，也可以用**重量**，或者用**容积**。所以大可在一个结构中创建多个字段：

```
typedef struct {
  ...
  short count;
  float weight;
  float volume;
  ...
} fruit;
```

这不是好主意，原因有以下几点：

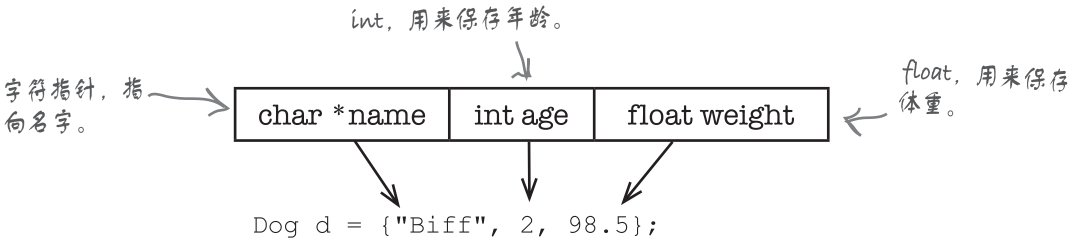
- 结构在存储器中占了更多空间。
- 用户可能设置多个值。
- 没有叫“量”的字段。

要是能这样就好了：定义一种叫“量”的数据类型，然后根据特定的数据决定要保存个数、重量还是容积。

在C语言中，可以用联合做到这点。

联合可以有效使用存储器空间

每次创建结构实例，计算机都会在存储器中相继摆放字段：



联合则不同。当定义联合时，计算机只为其中一个字段分配空间。假设你有一个叫quantity的联合，它有三个字段，分别是count、weight和volume，那么计算机就会为其中最大的字段分配空间，然后由你决定里面保存什么值。无论设置了count、weight和volume中的哪个字段，数据都会保存在存储器中同一个地方。

“量”（可能是float，也可能是short）

联合看起来很像结构，但用的是union关键字。

如果float占4字节，short占2字节，那么quantity将占4字节。

```
typedef union {
  short count;
  float weight;
  float volume;
} quantity;
```

所有字段将保存在同一个地方。

数据类型不同，但都表示“量”。

橙子的个数。

葡萄的重量。

果汁的容积。

如何使用联合

声明联合变量后，有很多方法设置它的值。

C89方式

如果联合要保存**第一个字段**的值，就可以用C89表示法，只要用花括号把值括起来，就可以把值赋给联合中第一个字段。

```
quantity q = {4};
```
表示“量”是4个。

指定初始化器

指定初始化器（designated initializer）按名设置联合字段的值：

```
quantity q = {.weight=1.5};
```
把联合设为浮点型的重量值。

“点”表示法

第三种设置联合值的方法是在第一行创建变量，然后在第二行设置字段的值。

```
quantity q;
q.volume = 3.7;
```

切记，无论用哪种方法设置联合的值，都只会保存一条数据。联合只是提供了一种让你创建支持不同数据类型的变量的方法。

这里没有蠢问题

问： 为什么联合的大小取决于最长的字段？

答： 计算机需要保证联合的大小固定。唯一的办法就是让它足够大，任何一个字段都能装得下。

问： 为什么C89表示法只能设置第一个字段？如果我传给联合`float`值，为什么不把它设为第一个`float`字段？

答： 这么做是为了避免歧义。假设你有一个`float`字段和一个`double`字段，那么计算机应该把`{2.1}`保存成`float`还是`double`呢？每次都把值保存在第一个字段中，你就知道数据是怎么初始化的。

C标准礼貌指南

可以用“指定初始化器”按名设置结构和联合字段，它属于C99标准。绝大多数现代编译器都支持“指定初始化器”，但如果你用的是C语言的变种，就要小心了，比如Objective C支持“指定初始化器”，但C++不支持。

看起来结构也能用“指定初始化器”，对吗？

是的，“指定初始化器”也可以用来设置结构字段的初值。
如果结构有很多字段，但你只想为其中某些字段设初值，“指定初始化器”就非常有用。同时它还能够提高代码的可读性：

```
typedef struct {
  const char *color;
  int gears;
  int height;
} bike;
bike b = {.height=17, .gears=21};
```

将设置height和gears字段，但不设置color字段。

联合常和结构一起用

创建联合相当于创建新的数据类型，也就是说可以在任何地方使用它的值，就像使用整型或结构那样的数据类型。例如，可以把联合和结构结合起来：

```
typedef struct {
  const char *name;
  const char *country;
  quantity amount;
} fruit_order;
```

可以用之前使用过的“点”表示法或“->”表示法访问“结构/联合”组合中的值：

这里是.amount，它是结构quantity类型字段的变量名。

这里连用了两次指定标识符。.amount用来初始化结构中的字段，.weight用来初始化.amount中的字段。

```
fruit_order apples = {"apples", "England", .amount.weight=4.2};
printf("This order contains %2.2f lbs of %s\n", apples.amount.weight, apples.name);
```

将打印“This order contains 4.20 lbs of apples”。

晕头转向的调酒师

Head First酒吧的“玛格丽特之夜”，一群人喝得酩酊大醉以后把配方弄乱了，你能否为每种玛格丽特酒找到相应的代码？

基本原料如下：

```c
typedef union {
  float lemon;
  int lime_pieces;
} lemon_lime;

typedef struct {
  float tequila;
  float cointreau;
  lemon_lime citrus;
} margarita;
```

下面是几种不同的玛格丽特酒：

```c
margarita m = {2.0, 1.0, {0.5}};
```

```c
margarita m = {2.0, 1.0, .citrus.lemon=2};
```

```c
margarita m = {2.0, 1.0, 0.5};
```

```c
margarita m = {2.0, 1.0, {.lime_pieces=1}};
```

```c
margarita m = {2.0, 1.0, {1}};
```

```c
margarita m = {2.0, 1.0, {2}};
```

最后，这里有一些不同的调法和他们制作的配方。为了生成正确的配方，应该把哪些玛格丽特酒加入代码？

```
....................................................................................................
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f
    measures of juice\n", m.tequila, m.cointreau, m.citrus.lemon);

2.0 measures of tequila
1.0 measures of cointreau
2.0 measures of juice
```

```
....................................................................................................
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f
    measures of juice\n", m.tequila, m.cointreau, m.citrus.lemon);

2.0 measures of tequila
1.0 measures of cointreau
0.5 measures of juice
```

```
....................................................................................................
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%i pieces
    of lime\n", m.tequila, m.cointreau, m.citrus.lime_pieces);

2.0 measures of tequila
1.0 measures of cointreau
1 pieces of lime
```

变身编译器

有的代码能编译，有的不能。你的任务是扮演编译器，说出哪段代码能编译，如果不能请说明理由。

```
margarita m = {2.0, 1.0, {0.5}};
```

```
margarita m;
m = {2.0, 1.0, {0.5}};
```

晕头转向的调酒师解答

Head First酒吧的“玛格丽特之夜”，一群人喝得酩酊大醉以后把配方弄乱了，你将为每种玛格丽特酒找到相应的代码。

基本原料如下：

```
typedef union {
  float lemon;
  int lime_pieces;
} lemon_lime;

typedef struct {
  float tequila;
  float cointreau;
  lemon_lime citrus;
} margarita;
```

下面是几种不同的玛格丽特酒：

```
margarita m = {2.0, 1.0, .citrus.lemon=2};
```

```
margarita m = {2.0, 1.0, 0.5};
```

没用到这几行代码。

```
margarita m = {2.0, 1.0, {1}};
```

最后，这里有一些不同的调法和他们制作的配方。为了生成正确的配方，应该把哪些玛格丽特酒加入代码？

margarita m = {2.0, 1.0, {2}};

```
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f
    measures of juice\n", m.tequila, m.cointreau, m.citrus.lemon);
```

```
2.0 measures of tequila
1.0 measures of cointreau
2.0 measures of juice
```

margarita m = {2.0, 1.0, {0.5}};

```
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%2.1f
    measures of juice\n", m.tequila, m.cointreau, m.citrus.lemon);
```

```
2.0 measures of tequila
1.0 measures of cointreau
0.5 measures of juice
```

margarita m = {2.0, 1.0, {.lime_pieces=1}};

```
printf("%2.1f measures of tequila\n%2.1f measures of cointreau\n%i pieces
    of lime\n", m.tequila, m.cointreau, m.citrus.lime_pieces);
```

```
2.0 measures of tequila
1.0 measures of cointreau
1 pieces of lime
```

变身编译器解答

有的代码能编译，有的不能。你的任务是扮演编译器，说出哪段代码能编译，如果不能请说明理由。

```
margarita m = {2.0, 1.0, {0.5}};
```

成功编译，恰好是以上鸡尾酒中的一种！

```
margarita m;
m = {2.0, 1.0, {0.5}};
```

这段代码不能编译，因为只有把{2.0, 1.0, {0.5}}和结构声明写在一行里，编译器才知道它代表结构，否则，编译器会认为它是数组。

喂，等等，你可以在联合中保存任何字段的值，这些不同类型的值保存在存储器中相同的位置……既然如此，你怎么知道我保存的是float还是short？要是我保存了float字段，却读取了short字段呢？

好问题：可以在联合中保存各种可能的值，但保存以后，就无法知道它的类型。

编译器不会记录你在联合中设置或读取过哪些字段。我们完全可以设置一个字段，读取另一个字段，但有时这会造成很严重的后果。

```
#include <stdio.h>
typedef union {
  float weight;
  int count;
} cupcake;

int main()
{
  cupcake order = {2};
  printf("Cupcakes quantity: %i\n", order.count);
  return 0;
}
```

程序员无意中设置了weight，而不是count。

明明设置了weight，却读取了count。

程序的运行结果如下。

```
File Edit Window Help
> gcc badunion.c -o badunion && ./badunion
Cupcakes quantity: 1073741824
```

数不清的纸托蛋糕……

你需要某种方法记录我们在联合中保存了什么值。C程序员常用的一种技巧是创建**枚举**。

枚举变量保存符号

有时你不想保存数字或文本，而是想保存一组符号。如果你想记录一周中的某一天，只想保存MONDAY、TUESDAY、WEDNESDAY……这些符号。你不需要保存文本，因为一共只有七种不同的取值。

这就是为什么要发明枚举的原因。

有了枚举，就可以创建一组这样的符号：

枚举中所有可能的颜色。

值以逗号分隔。

```
enum colors {RED, GREEN, PUCE};
```

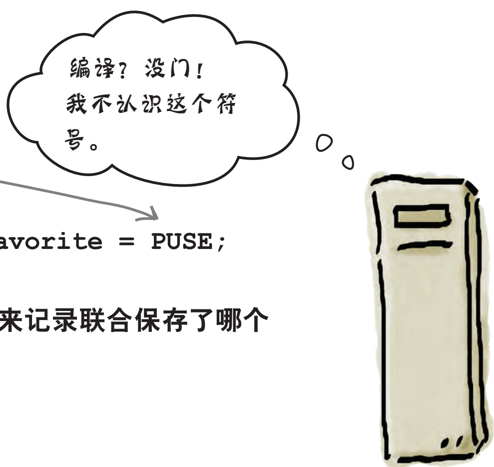
可以用typedef为类型起个名字。

用**enum colors**类型定义的变量只能设为列表中的某个关键字。可以像下面这样定义enum colors变量：

```
enum colors favorite = PUCE;
```

结构与联合用分号（;）来分割数据项，而枚举用逗号。

在幕后，计算机会为列表中的每个符号分配一个数字，枚举变量中也只保存数字。至于是什么数字，你完全不用操心，C程序员只要在代码中写符号就行了。枚举不仅让代码更易于阅读，同时它也能够防止你把值保存成REB或PUSE：



计算机会指出PUSE是非法值，因此不会编译代码。

```
enum colors favorite = PUSE;
```

枚举就是这么工作的，但如何用它来记录联合保存了哪个字段呢？来看一个例子……

代码冰箱贴

你既然能用枚举创建新的数据类型，也就能把它保存在结构或联合中。下面这个程序用枚举变量来记录“量”的类型，你能找到丢失的代码吗？

```
#include <stdio.h>

typedef enum {
  COUNT, POUNDS, PINTS
} unit_of_measure;

typedef union {
  short count;
  float weight;
  float volume;
} quantity;

typedef struct {
  const char *name;
  const char *country;
  quantity amount;
  unit_of_measure units;
} fruit_order;

void display(fruit_order order)
{
  printf("This order contains ");

  if (........................== PINTS)

    printf("%2.2f pints of %s\n", order.amount.........................., order.name);
```

```
  else if (.............. == ..............)
    printf("%2.2f lbs of %s\n", order.amount.weight, order.name);
  else

    printf("%i %s\n", order.amount. .............. , order.name);
}

int main()
{

  fruit_order apples = {"apples", "England", .amount.count=144, ..............};

  fruit_order strawberries = {"strawberries", "Spain", .amount...........=17.6, POUNDS};

  fruit_order oj = {"orange juice", "U.S.A.", .amount.volume=10.5, ..............};
  display(apples);
  display(strawberries);
  display(oj);
  return 0;
}
```

POUNDS

order.units

COUNT

weight

volume

count

PINTS

order.units

代码冰箱贴解答

你既然能用枚举创建新的数据类型，也就能把它保存在结构或联合中。下面这个程序用枚举变量来记录“量”的类型，你找到丢失的代码了吗？

```
#include <stdio.h>

typedef enum {
  COUNT, POUNDS, PINTS
} unit_of_measure;

typedef union {
  short count;
  float weight;
  float volume;
} quantity;

typedef struct {
  const char *name;
  const char *country;
  quantity amount;
  unit_of_measure units;
} fruit_order;

void display(fruit_order order)
{
  printf("This order contains ");

  if (order.units == PINTS)

    printf("%2.2f pints of %s\n", order.amount.volume, order.name);
```

```
  else if (order.units == POUNDS)
    printf("%2.2f lbs of %s\n", order.amount.weight, order.name);
  else

    printf("%i %s\n", order.amount.count, order.name);
}

int main()
{

  fruit_order apples = {"apples", "England", .amount.count=144, COUNT};

  fruit_order strawberries = {"strawberries", "Spain", .amount.weight=17.6, POUNDS};

  fruit_order oj = {"orange juice", "U.S.A.", .amount.volume=10.5, PINTS};
  display(apples);
  display(strawberries);
  display(oj);
  return 0;
}
```

运行程序，将得到：

```
File Edit Window Help
> gcc enumtest.c -o enumtest && ./enumtest
This order contains 144 apples
This order contains 17.60 lbs of strawberries
This order contains 10.50 pints of orange juice
```

联合：……于是我对代码说，“瞧，不管你给我的是不是float，只要你问我要int，我就给你int。”

结构：哥们，这么做完全没有道理。

联合：就是没有道理。

结构：人人都知道你只有一块存储空间。

联合：是的，万物一体，佛曰……

枚举：发生了什么？

结构：枚举，闭嘴，这个家伙太过分了。

联合：如果用户只给出一条记录，比如在这个例子中，我就把它保存成int，只要用枚举或其他东西记录一下就行了。

枚举：你想让我来做这件事？

结构：闭嘴，枚举。

联合：如果用户希望一次保存多条数据，就应该用你，对吗？

结构：这群人根本不知道什么是秩序。

枚举：什么是秩序？

结构：秩序意味着独立且有序。我在存储器中相继摆放数据，同时保存所有数据。

联合：不错。

结构：同时保存所有数据（字正腔圆）。

枚举：（停顿）有问题吗？

联合：小声点，枚举。我认为这需要由用户来决定，如果他想保存多条数据，就用你；如果想保存有多种类型的数据，就选我。

结构：我给他打个电话。

联合：喂，等等……

枚举：他在给谁打电话，哥们？

结构/联合：闭嘴，枚举。

联合：别惹麻烦了，行吗？

结构：喂？请帮我接蓝牙服务。

联合：喂，我们再考虑一下。

结构：什么？让他过会儿给我回电话？

联合：也许这不是一个好主意。

结构：不用，帮我传句话，朋友。

联合：求你把电话挂掉。

枚举：电话那头是谁？

结构：小声点，枚举，没看见我在打电话么？听好了，你只要告诉他说，如果他想保存float和int变量就来找我，要么我去找他，明白了吗？喂？喂？

联合：沉住气，冷静。

结构：我X，转接中！

联合：怎么了？把电话给我……不……电话里在放老鹰乐队的歌！我讨厌老鹰乐队……

枚举：你之所以那么胖，就是因为把字段打包在了一起？

结构：朋友，我们到外面聊两句。

有时你想控制某一位

假设你需要一个结构，其中有很多表示“是”或“非”的值，可以用一些short或int来创建结构：

```
typedef struct {
  short low_pass_vcf;
  short filter_coupler;
  short reverb;
  short sequential;
  ...
} synth;
```

这些字段要么保存1（表示真），要么保存0（表示假）。

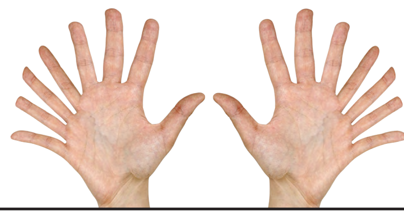
后面还有很多字段。

每个字段都会使用许多位。

0000000000000001	0000000000000001	0000000000000001	...

这样做完全可行，但问题是，真/假的值只需要一位就能表示，short字段占了多得多的空间，太浪费了，要是结构的字段的值能只用一位表示就好了。

这就是发明位字段（**bitfield**）的原因。



二进制位百宝箱

处理二进制值时，要是能够以某种方法在字面值中指定0和1就好了，比如：

```
int x = 01010100;
```

可惜的是，C语言不支持**二进制字面值**，不过它支持**十六进制字面值**。每当C语言看到0x开头的数字，就认为它是以**16为基数**的数字：

```
int x = 0x54;
```

不是十进制数54。

如何在十六进制与二进制之间进行转换？这比二进制与十进制之间的转换要容易些吗？是的，可以把十六进制数逐位转换为二进制数：

```
        0x54
       /    \
   0101      0100
```

这是5。 这是4。

每一个十六进制数对应一个长度为4的二进制数。只要知道数字0到15的二进制形式，就能很快地在心中完成转换。

位字段的位数可调

可以用位字段（bitfield）指定一个字段有多少位。例如，可以把结构写成这样：

```
typedef struct {
  unsigned int low_pass_vcf:1;
  unsigned int filter_coupler:1;
  unsigned int reverb:1;
  unsigned int sequential:1;
  ...
} synth;
```

每个字段都必须是unsigned int。

表示该字段只使用1位存储空间。

使用位字段，就可以保证每个字段只占1位。

1	1	1	...

如果你有一连串的位字段，计算机会**放在一起**，以节省空间，也就是说如果有8个1位的位字段，计算机就会把它们保存在一个字节中。

让我们看看如何巧用位字段。

只有出现在同一个结构中，位字段才能节省空间。

如果编译器发现结构中只有一个位字段，还是会把它填充成一个字，这就是为什么位字段总是组合在一起。

如何选择位数？

位字段不仅可以用来保存一连串真/假值，还可以用来保存小范围的数字，例如一年中的十二个月。假设想在某个结构中保存月份（0到11的值），就可以用一个**4位**的位字段来保存，为什么？因为4位可以保存0到15，而3位只能保存0到7。

```
...
unsigned int month_no:4;
...
```

练习

回到Head First水族馆，工作人员正在对顾客进行满意度调查，你能否利用位字段创建相应的结构？

Head First 水族馆

水族馆调查问卷

您是第一次参观水族馆吗？	
您还会再来吗？	
您被食人鱼咬掉几根手指？	
您的小孩是否在鲨鱼表演时遇难？	
如果可以的话，您一周会来参观几次？	

```
typedef struct {
  unsigned int first_visit: .......... ;
  unsigned int come_again: .......... ;
  unsigned int fingers_lost: .......... ;
  unsigned int shark_attack: .......... ;
  unsigned int days_a_week: .......... ;
} survey;
```

你需要决定使用多少位。

练习解答

回到Head First水族馆，工作人员正在对顾客进行满意度调查，请利用位字段创建相应的结构。

Head First 水族馆

水族馆调查问卷

您是第一次参观水族馆吗？	
您还会再来吗？	
您被食人鱼咬掉几根手指？	
您的小孩是否在鲨鱼表演时遇难？	
如果可以的话，您一周会来参观几次？	

```
typedef struct {
  unsigned int first_visit: 1 ;
  unsigned int come_again: 1 ;
  unsigned int fingers_lost: 4 ;
  unsigned int shark_attack: 1 ;
  unsigned int days_a_week: 3 ;
} survey;
```

1位可以保存2个值：真或假。

保存0到10的值，需要4位。

3位可以保存0到7的值。

这里没有蠢问题

问： 为什么C语言不支持二进制字面值？

答： 因为二进制字面值占了很大空间，而且十六进制通常写起来更快。

问： 为什么保存一个0到10的值需要4位？

答： 4位可以保存0到二进制数1111（也就是15）的值，但3位最大只能保存二进制数111（也就是7）。

问： 如果我把9放到一个3位的字段中会怎样？

答： 计算机会保存1，因为9的二进制是1001，所以计算机转换成001。

问： 位字段就是为了节省空间的吗？

答： 不仅仅是为了节省空间，如果需要读取低层的二进制信息，位字段就会非常有用。

问： 能举个例子吗？

答： 比如要读写某类自定义二进制文件。

要点

- 可以用联合在同一个存储器单元中保存不同数据类型。
- “指定初始化器”按名设置字段的值。
- C99标准支持“指定初始化器”，C++不支持。
- 如果用{花括号}中的值初始化联合，这个值会以第一个字段的类型保存。
- 你完全可以读取联合中未初始化过的字段，编译器不会干涉。但要小心，因为这么做很有可能会出错。
- 枚举保存符号。
- 可以用位字段自定义字段的位数。
- 位字段应当声明为`unsigned int`。

C语言工具箱

学完第5章，现在你的工具箱中又多出了结构、联合与位字段。关于本书提示工具条的完整列表，请见附录ii。

可以用typedef为数据类型创建别名。

结构把数据类型组合在一起。

可以用“点表示法”读取结构中的字段

可以像初始化数组那样初始化结构。

有了“->”表示法，就能用结构指针更新字段，十分方便。

可以用“指定初始化器”按名设置结构或联合的字段。

联合可以在同一个存储器单元中保存不同的数据类型。

可以用枚举创建一组符号。

可以用位字段控制结构中的某些位。

6 数据结构与动态存储

牵线搭桥

一个结构根本不够。

为了模拟复杂的数据需求，通常需要把结构链接在一起。在本章中，你将学习如何用结构指针把自定义数据类型连接成复杂的大型数据结构，将通过创建链表来探索其中的基本原理；同时还将通过在堆上动态地分配空间来学习如何让数据结构处理可变数量的数据，并在完成工作后释放空间；如果你嫌清理工作太麻烦，可以学习一下怎么用`valgrind`。

保存可变数量的数据

椰果航空的飞机在岛屿之间飞来飞去。

在C语言中，你可以保存不同类型的数据，也能在数组中保存多条数据，但有时需要更灵活一点。

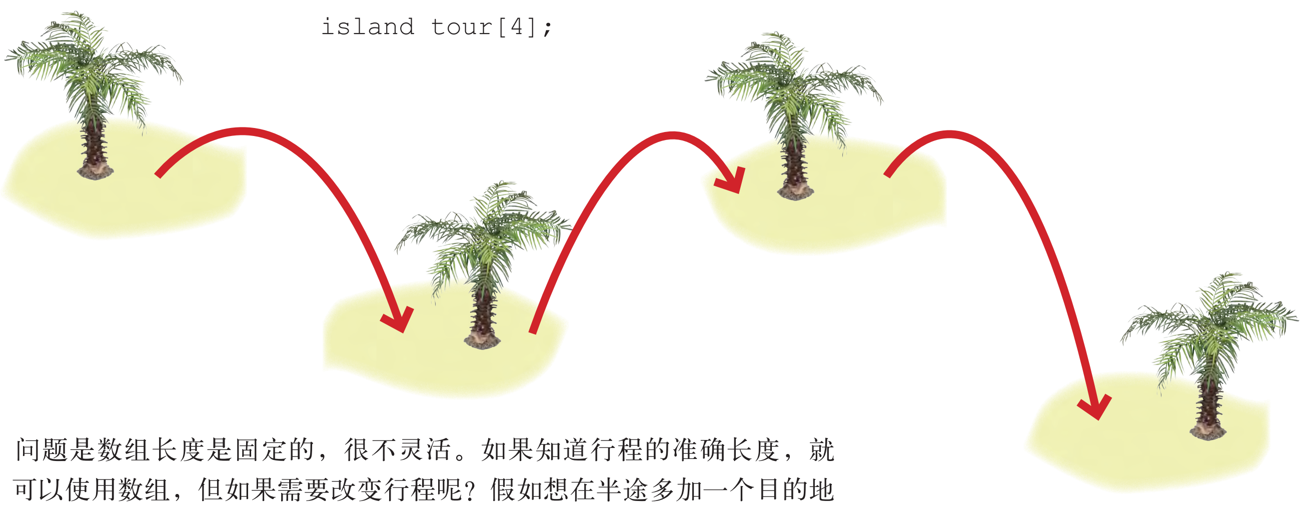
想象你正在经营一家旅行公司，专门策划跨岛的飞行之旅。行程包含一系列的短程航班，从一座岛到另一座岛。你需要记录每一座岛的信息，例如岛名和机场的营业时间，你准备怎么记录？

可以创建结构来表示一座岛：

```
typedef struct {
  char *name;
  char *opens;
  char *closes;
} island;
```

飞行之旅途经一连串的岛屿，也就是说需要记录一列岛，因此可以创建island数组。

```
island tour[4];
```



问题是数组长度是固定的，很不灵活。如果知道行程的准确长度，就可以使用数组，但如果需要改变行程呢？假如想在半途多加一个目的地呢？

为了保存可变数量的数据，需要一样比数组更灵活的东西，即链表。

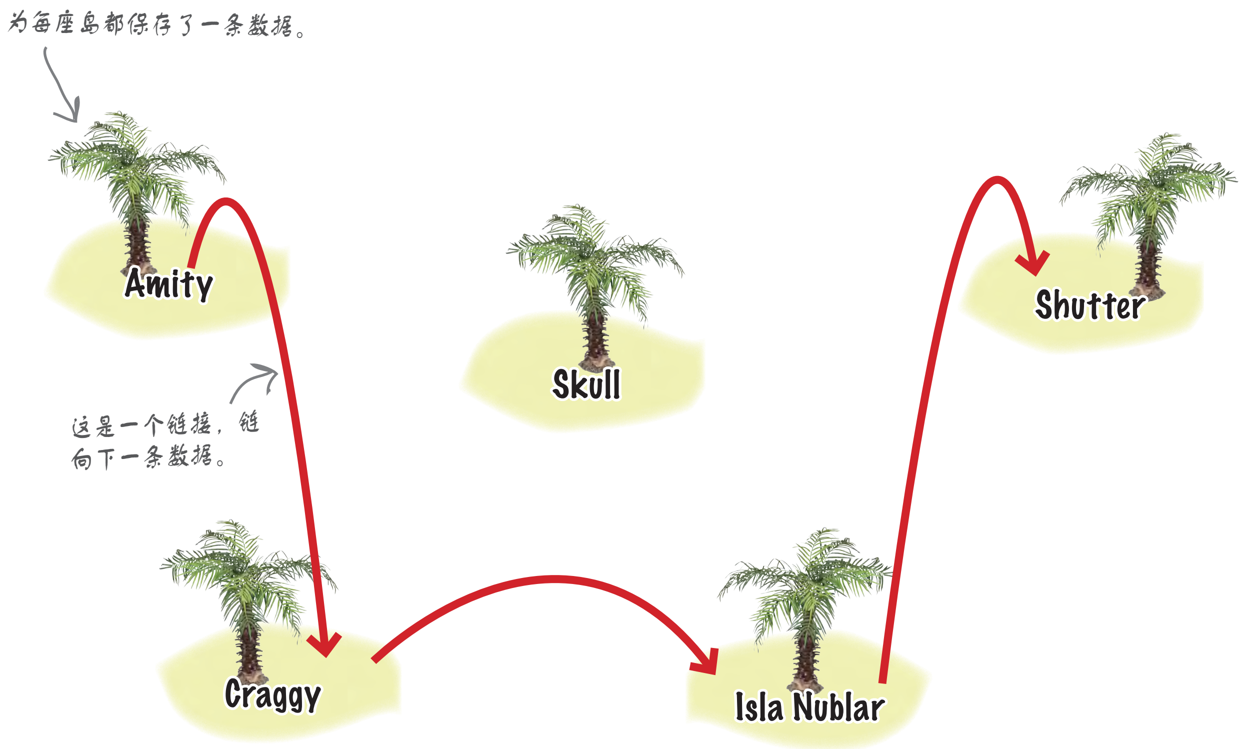
链表就是一连串的数据

链表是一种**抽象数据结构**。链表是通用的，可以用来保存很多不同类型的数据，所以被称之为抽象数据结构。

为了理解链表是怎么工作的，回想一下我们的旅行公司。链表保存了一条数据和**一个链向另一条数据的链接**。

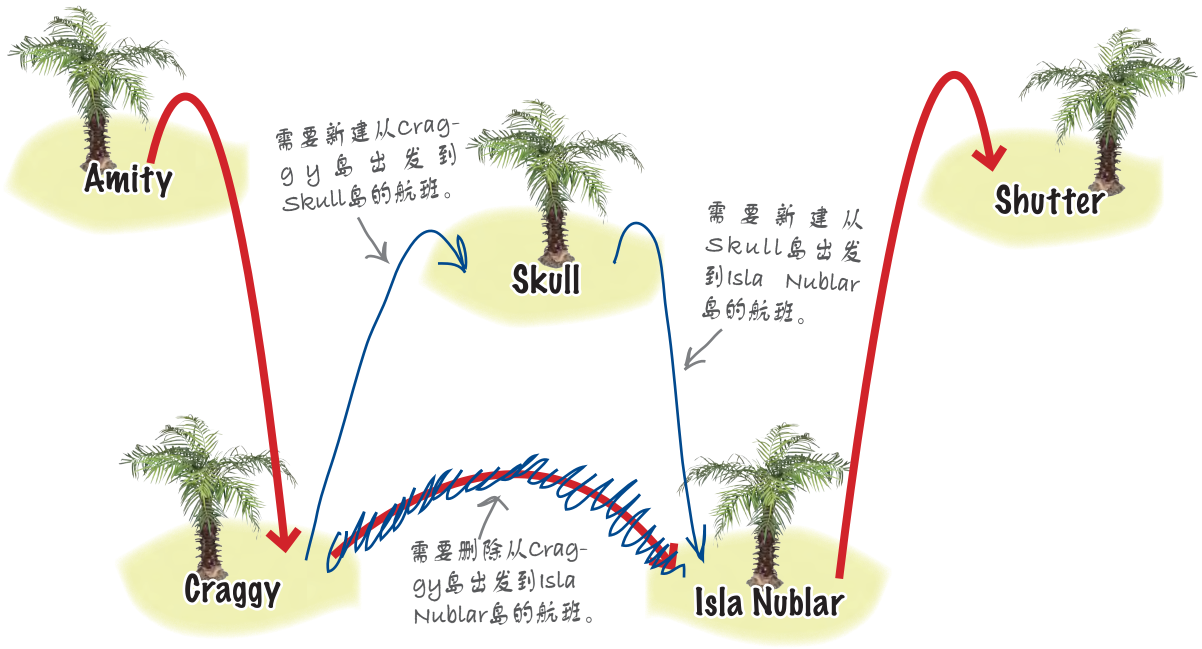
磨笔上阵

只要知道链表从哪里开始，就可以遍历链表。可以从一条数据跳到另一条，直到到达链表的尾部。请用铅笔修改链表，在Craggy岛与Isla Nublar岛之间增加一次到Skull岛的旅行。



磨笔上阵 解答

只要知道链表从哪里开始，就可以遍历链表。可以从一条数据跳到另一条，直到到达链表的尾部。请用铅笔修改了链表，在Craggy岛与Isla Nublar岛之间增加了一次到Skull岛的旅行。



在链表中插入数据

只要稍加改动，就在旅行中多加了一站。与数组相比，链表还有一个优点：插入数据非常快。如果想在数组中插入一个值，就不得不将插入点后所有数据后移一个位置：

如果想在Craggy岛后多插一个值，不得不将其他值后移一个位置。

这是数组。

Amity	Craggy	Isla Nublar	Shutter

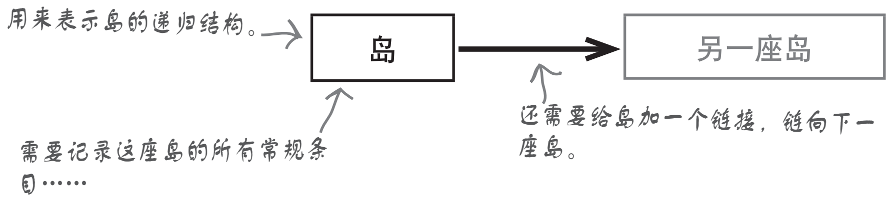
因为数组长度固定，你将失去Shutter岛。

你可以用链表来保存可变数量的数据，而且在链表中**添加数据**很简单。

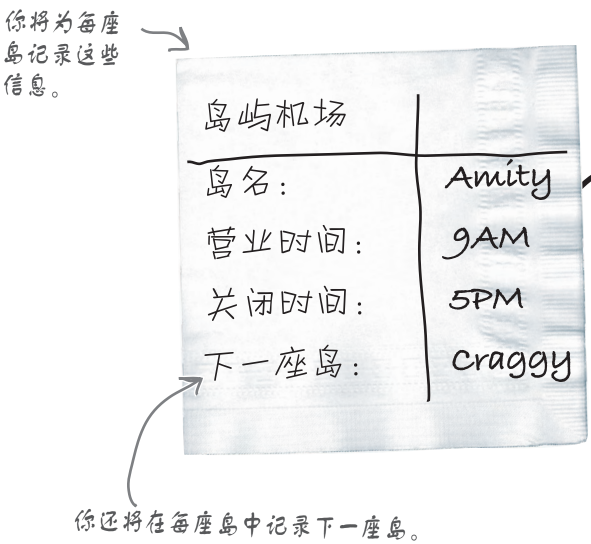
如何在C语言中创建链表？

创建递归结构

链表中的每个结构都需要与下一个结构相连。如果一个结构包含一个链向同种结构的链接，那么这个结构就被称为**递归结构**。



递归结构含有指向同种结构的指针。如果你有一张航班时间表，上面列出了将要游览的岛屿，就可以用递归结构表示island，下面具体讨论递归结构是怎么工作的：



你必须为这个结构命名。

```
typedef struct island {
   char *name;
   char *opens;
   char *closes;
   struct island  *next;
} island;
```

你将用字符串表示岛名和机场的营业时间。

你在结构中保存了一个指针，指向下一座岛。

如何在当前结构中保存链向下一个结构的链接呢？用指针。只要在结构中保存指针，island数据就含有下一个我们将游览的island的地址。只要我们的代码能访问一个island，就能够跳到下一个island。

下面开始写代码，开启我们的岛间飞行之旅。

递归结构要有名字。

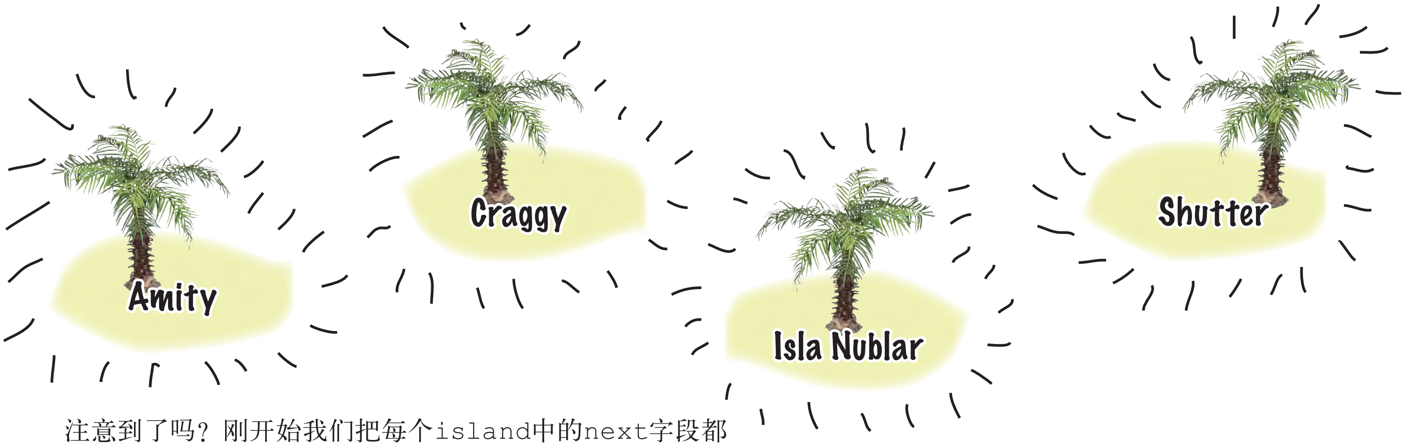
当用typedef命令定义结构时可以跳过为结构起名字这步，但在递归结构中，需要包含一个相同类型的指针，C语言的语法不允许用typedef别名来声明它，因此必须为结构起一个名字。这就是为什么这里的结构叫struct island。

用C语言创建岛屿……

一旦定义了island数据类型，就可以像这样创建第一批island：

这段代码将为每座岛创建island结构。

```
island amity = {"Amity", "09:00", "17:00", NULL};
island craggy = {"Craggy", "09:00", "17:00", NULL};
island isla_nublar = {"Isla Nublar", "09:00", "17:00", NULL};
island shutter = {"Shutter", "09:00", "17:00", NULL};
```

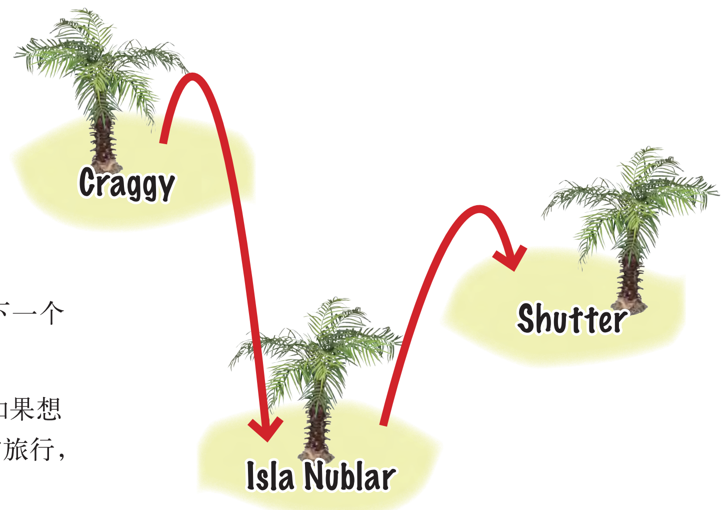


注意到了吗？刚开始我们把每个island中的next字段都设为了NULL。在C语言中，NULL的值实际上为0，NULL专门用来把某个指针设为0。

……把它们链接在一起，构成飞行之旅

一旦你创建好了岛，就可以把它们连接在一起：

```
amity.next = &craggy;
craggy.next = &isla_nublar;
isla_nublar.next = &shutter;
```



你必须小心地将每一个island的next字段设为下一个island的地址，你将使用每座岛的结构变量。

现在你已经用C语言创建了一次完整的跳岛游，但如果想在Isla Nublar岛与Shutter岛之间插入一次到Skull岛的旅行，该怎么做？

在链表中插入值

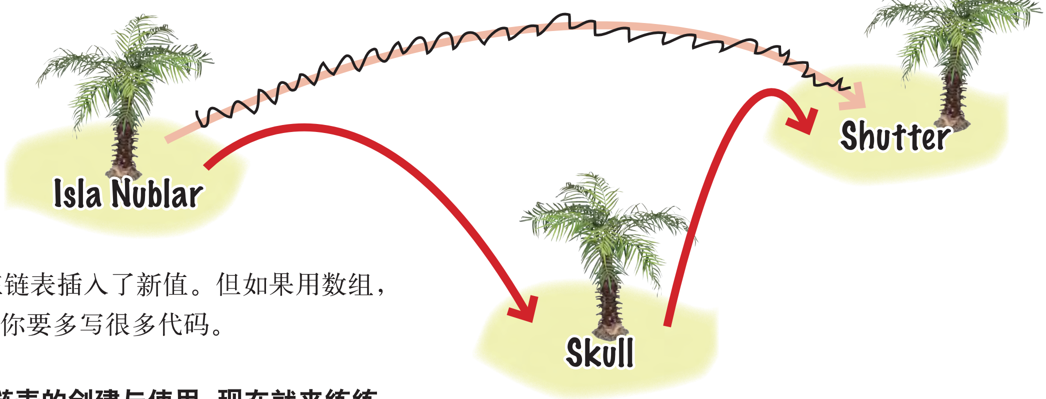
通过修改指针的值，就可以插入island，就像之前做的那样：

这行代码创建了Skull岛。

```
island skull = {"Skull", "09:00", "17:00", NULL};
isla_nublar.next = &skull;
skull.next = &shutter;
```

这行代码把Isla Nublar岛连到Skull岛。

这行代码把Skull岛连到Shutter岛。



短短两行代码，就在链表插入了新值。但如果用数组，为了移动数组元素，你要多写很多代码。

好了，你已经学会了链表的创建与使用，现在就来练练吧……

代码冰箱贴

不好啦，有人弄乱了冰箱门上的display()函数，你能重组代码吗？

```
void display(island *start)
{
  island *i = start;

  for (; i ........ ........ ; i ........ ........ ) {

    printf("Name: %s\n open: %s-%s\n", ........ , ........ , ........ );
  }
}
```

i->closes　　!=　　i->opens　　i->name　　NULL　　i->next　　=

代码冰箱贴解答

不好啦，有人弄乱了冰箱门上的display()函数，你重组代码了吗？

```
void display(island *start)
{
  island *i = start;

  for (; i != NULL ; i = i->next ) {

    printf("Name: %s\n open: %s-%s\n", i->name, i->opens, i->closes );
  }
}
```

无需在循环开始时添加额外的代码。

需要一直循环下去，直到当前island没有next值。

在每次循环的最后，跳到下一座岛。

这里没有蠢问题

问： 其他语言，比如Java，有内置链表，C语言有内置数据结构吗？

答： C语言没有内置数据结构，你必须自己创建它们。

问： 要是我有一个很长的链表，如果我想使用第700个元素，就必须从第一个开始一路读下去吗？

答： 是的，你必须这样做。

问： 这样可不好，本来我以为链表比数组好。

答： 数据结构没有好与坏之分，只有适合或不适合于它的应用场合之分。

问： 也就是说如果我想快速地插入数据，就需要链表，但如果我想直接访问元素，就应该用数组。是这样吗？

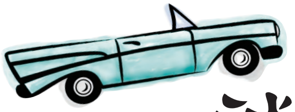
答： 完全正确。

问： 你给出的这个结构含有一个指向其他结构的指针。我能把指针换成一个递归定义的结构吗？

答： 不行。

问： 为什么？

答： C语言需要知道结构在存储器中占的具体大小，如果在结构中递归地复制它自己，那么两条数据就会不一样大。



试驾

我们对island链表使用display()函数，并把代码编译成一个叫tour的程序。

```
island amity = {"Amity", "09:00", "17:00", NULL};
island craggy = {"Craggy", "09:00", "17:00", NULL};
island isla_nublar = {"Isla Nublar", "09:00", "17:00", NULL};
island shutter = {"Shutter", "09:00", "17:00", NULL};
amity.next = &craggy;
craggy.next = &isla_nublar;
isla_nublar.next = &shutter;
island skull = {"Skull", "09:00", "17:00", NULL};
isla_nublar.next = &skull;
skull.next = &shutter;
display(&amity);
```

File Edit Window Help GetBiggerBoat

```
> gcc tour.c -o tour && ./tour
Name: Amity
Open: 09:00-17:00
Name: Craggy
Open: 09:00-17:00
Name: Isla Nublar
Open: 09:00-17:00
Name: Skull
Open: 09:00-17:00
Name: Shutter
Open: 09:00-17:00
>
```

妙极了，代码创建了island链表，还能方便地插入元素。

好了，现在你已经掌握了使用递归结构和链表的基本方法，现在来看主程序。你需要从下面这个文件中读取飞行之旅的数据：

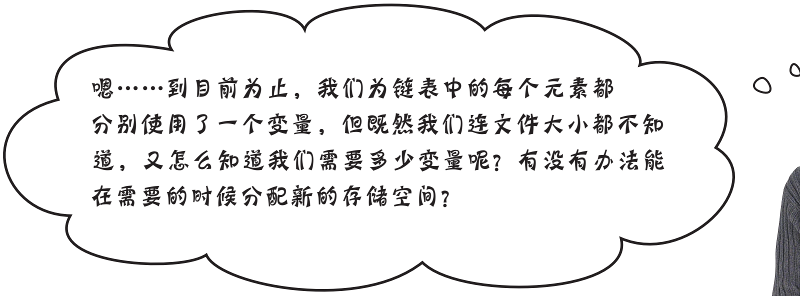
```
Delfino Isle
Angel Island
Wild Cat Island
Neri's Island
Great Todday
```

下面还有很多行。

机场的工作人员还在创建文件，在程序停止之前，你都不知道文件有多大。文件中每一行都是一个岛名，把文件转换为链表应该不难，你说呢？

C标准礼貌指南

本页上的代码在中间的位置声明了新变量skull，只有C99和C11标准才允许这样做，在ANSI C中，必须在函数的顶部声明局部变量。



没错，需要以某种方法创建动态存储。

到目前为止你写过的所有程序都使用了静态存储。每当你想保存一样东西，都在代码中添加了一个变量。这些变量通常保存在栈中，别忘了，栈是存储器中专门用来保存局部变量的区域。

当你创建前四座岛时，写了：

```
island amity = {"Amity", "09:00", "17:00", NULL};
island craggy = {"Craggy", "09:00", "17:00", NULL};
island isla_nublar = {"Isla Nublar", "09:00", "17:00", NULL};
island shutter = {"Shutter", "09:00", "17:00", NULL};
```

每个`island`结构都需要自己的变量。上面这段代码始终只会创建这四座岛。如果想要让代码保存更多的`island`，需要使用更多的局部变量。如果在编译时知道需要保存多少数据，那没问题，但程序在运行前往往不知道自己需要多少存储空间。打个比方，假如你在编写网页浏览器，那么在读取网页之前就不知道保存网页需要多少存储空间。因此C程序需要以某种方式让操作系统在它们需要的时候分配存储空间。

程序需要动态存储。

要是我能在程序运行时分配空间那该多好，但我知道这是痴心妄想……

用堆进行动态存储

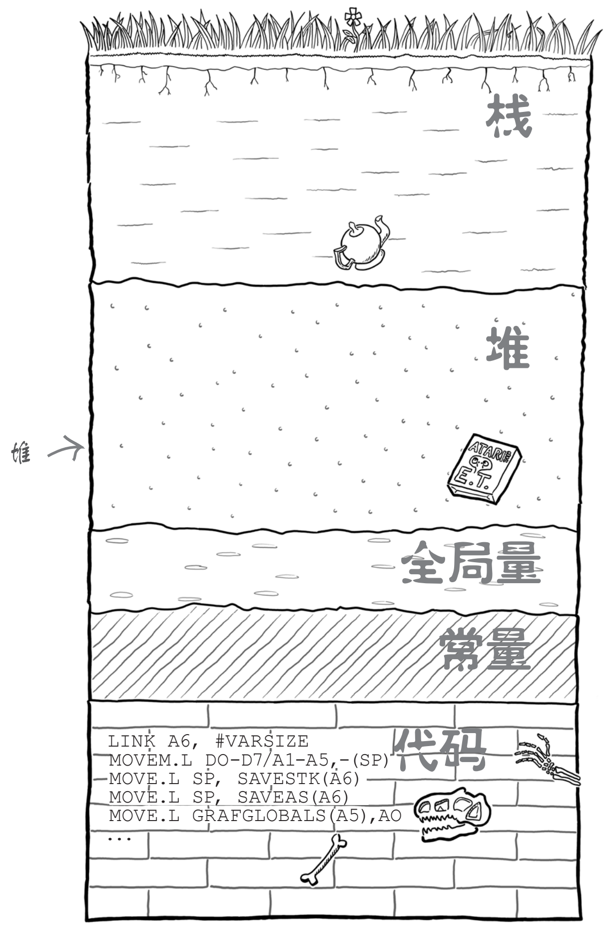
目前你用过的大部分存储器都在**栈**上。栈是存储器用来保存局部变量的区域。数据保存在局部变量中，一旦离开函数，变量就会消失。

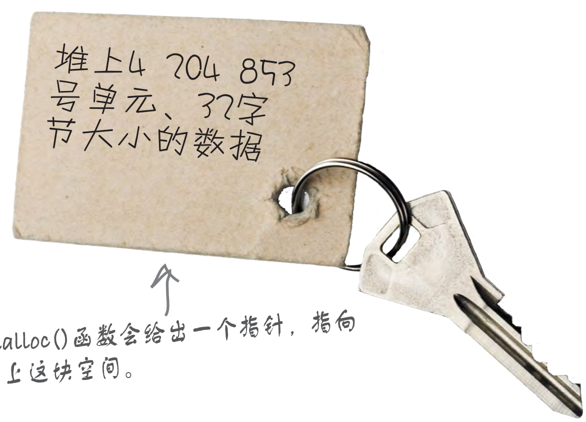
问题是程序在运行时很难在栈上分配大量空间，所以你需要**堆**。堆是程序中用来保存长期使用数据的地方。堆上的数据不会自动清除，因此堆是保存数据结构的绝佳场所，比如我们的链表。可以把在堆上保存数据想象成在储物柜中寄存物品。

在堆中保存数据就好比在储物柜中保存变量。

首先，用malloc()获取空间

想象程序在运行时突然发现有大量数据要保存，程序想申请一个大容量储物柜来保存数据，在C语言中，可以用一个叫**malloc()**的函数来申请。你告诉malloc()需要多少存储器，它就会要求操作系统在堆中分配这点空间，然后malloc()会返回一个**指针**，指向堆上新分配的空间。指针就好比储物柜的钥匙，可以用它来访问存储器，并跟踪这个分配出去的储物柜。





malloc()函数会给出一个指针，指向堆上这块空间。

有用有还

堆存储器的优点就是可以占用它很长一段时间，缺点还是……可以占用它非常长的时间。

使用栈的时候，你无需操心归还存储器，因为这个过程是自动进行的。每当你离开函数，局部变量就会从栈中清除。

但堆完全不一样。一旦申请了堆上的空间，这块空间就再也不能分配出去，直到告诉C标准库你已经用完了。堆存储器的空间有限，如果在代码中不断地申请堆空间，很快就会发生存储器泄漏。

当程序不断地申请存储器，又不释放那些不再需要的存储器，就会发生存储器泄漏。存储器泄漏是C程序中最常见的错误，它们很难追踪。

堆空间有限，请合理使用。

调用free()释放存储器

malloc()函数分配空间并给出一个指向这块空间的指针。你需要用这个指针访问数据，用完以后，需要用**free()**函数释放存储器，就像把储物柜的钥匙还给服务员，好让别人能接着用。

每次在代码中用malloc()函数请求堆存储，就应该有相应的代码用free()函数归还存储空间。虽然程序结束以后，所有堆空间会自动释放，但用free()显式释放你创建的所有动态存储器是一种好的做法。

看看malloc()和free()如何工作。

用malloc()申请存储器……

申请存储器的函数叫malloc()，是memory allocation（存储器分配）的意思。malloc()接收一个参数：所需要的字节数。通常你不知道确切的字节数，因此malloc()经常与sizeof运算符一起使用，像这样：

```
#include <stdlib.h>
...
malloc(sizeof(island));
```

为了使用malloc()和free()函数，需要包含stdlib.h头文件。

表示“给我足够的空间来保存island结构”。

sizeof告知某种数据类型在系统中占了多少字节。这种数据类型可以是结构，也可以是int或double这样的基本数据类型。

malloc()函数为你分配一块存储器，然后返回一个指针，指针中保存了存储器块的起始地址。那么这个指针是什么类型呢？事实上，malloc()返回的是通用指针，即**void***类型的指针。

```
island *p = malloc(sizeof(island));
```

表示“为island创建足够空间，然后将地址保存在变量p中。

……用free()释放存储器

一旦在堆上创建了存储器，就可以随时使用它。一旦完成了工作，就需要用free()函数释放存储器。

free()需要接收malloc()创建的存储器的地址。只要告诉C标准库存储器块从哪里开始，它就能查阅记录，知道要释放多少存储器。假如想要释放上面那行代码分配的存储器，可以这样做：

```
free(p);
```

表示“释放你分配的存储器，从堆地址p开始”。

记住：如果在一个地方用malloc()分配了存储器，就应该在后面用free()释放它。

好了，现在我们对动态存储器有了更深的了解，可以开始写代码了。

不好！兼职演员来了……

这群有抱负的演员又有一部戏杀青了，所以他们有时间来帮你写代码。他们写了一个实用函数，函数根据你传给它的名字创建新的island结构：

```
island* create(char *name)
{
  island *i = malloc(sizeof(island));
  i->name = name;
  i->opens = "09:00";
  i->closes = "17:00";
  i->next = NULL;
  return i;
}
```

这是新函数。

岛名以字符指针的形式传递。

在堆上创建新的island结构。

用malloc()函数在堆上创建空间。

这几行代码设置了新结构的字段。

sizeof运算符计算出island结构需要多少字节。

函数返回了新结构的地址。

这个函数看起来很酷。演员们发现岛上大部分机场的开关门时间相同，所以他们把open和close字段设为了默认值，函数返回一个指针，指向新创建结构。

脑力风暴

仔细观察create()函数的代码，你认为会有问题吗？好好想一想，然后再翻页。

消失的岛屿案件

航空日志：11:00，周五，晴。我们编写了`create()`函数，它动态分配存储器。软件组的人说它可以用于试飞。

```
island* create(char *name)
{
  island *i = malloc(sizeof(island));
  i->name = name;
  i->opens = "09:00";
  i->closes = "17:00";
  i->next = NULL;
  return i;
}
```

14:15，多云，百慕大附近，方向西北，每小时15海里，逆风飞行。我们在第一站降落，软件组提供了基本的代码，我们在命令行中输入了岛名。

创建数组，保存岛名。→

要求用户输入岛名。→

```
char name[80];
fgets(name, 80, stdin);
island *p_island0 = create(name);
```

```
File Edit Window Help
> ./test_flight
Atlantis
```

14:45，发生地震，飞机在起飞时发生了轻微的摇晃。软件组在飞机上待命，可乐供给不足。

15:35，到达第二座岛，天气晴朗，无风。在程序中输入信息。

要求用户输入第二座岛的名字。

```
fgets(name, 80, stdin);
island *p_island1 = create(name);
p_island0->next = p_island1;
```

创建第二座岛。

把第一座岛与第二座岛连接起来。

```
File Edit Window Help
Titchmarsh Island
```

17:50，返回总部，整理日志。奇怪的事发生了，测试程序生成的飞行日志似乎出了错。程序记录了今日的行程，但第一座岛的名字莫名其妙地修改了，赶快让软件组来调查此事。

使用我们之前创建的函数显示island链表的内容。

```
display(p_island0);
```

```
File Edit Window Help
Name: Titchmarsh Island
 open: 09:00-17:00
Name: Titchmarsh Island
 open: 09:00-17:00
```

Atlantis去哪了？？？

第一座岛的名字竟然和第二座岛相同！！！

第一座岛的名字怎么了？`create()`函数出错了吗？你能从调用函数的代码中看出端倪吗？

消失的岛屿案件

第一座岛的名字怎么了？

再看一遍create()函数：

```
island* create(char *name)
{
  island *i = malloc(sizeof(island));
  i->name = name;
  i->opens = "09:00";
  i->closes = "17:00";
  i->next = NULL;
  return i;
}
```

当代码记录岛名时，并没有接收一份完整的name字符串，而只是记录了name字符串在存储器中的地址。这有关系吗？name字符串在哪里？为了回答这两个问题，我们看一眼调用create()函数的代码。

```
char name[80];
 fgets(name, 80, stdin);
 island *p_island0 = create(name);
 fgets(name, 80, stdin);
 island *p_island1 = create(name);
```

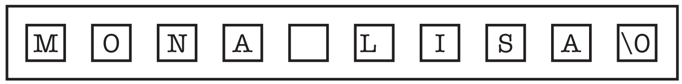
程序要求用户输入每座岛的名字，但两次它都使用本地的字符数组name来保存岛名，也就是说**两座岛共享**同一个name字符串，一旦局部变量name更新为第二座岛的名字，第一座岛的名字也就改变了。

聚焦字符串复制

在C语言中，经常需要复制字符串。你可以调用`malloc()`函数在堆上创建一些空间，然后手动把字符串的所有字符复制到堆上。但你猜怎么着？其他程序员早就想到了，他们在*string.h*头文件中创建了一个叫**strdup()**的函数。

假设你有一个指向你想复制的字符串常量的指针：

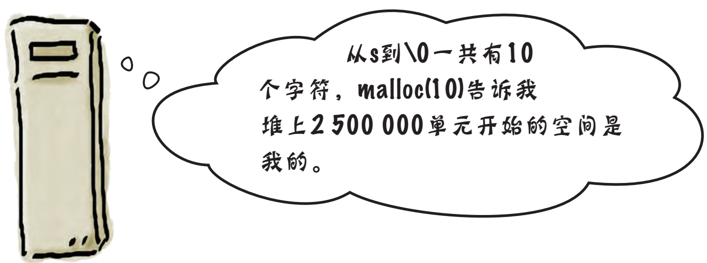
```
char *s = "MONA LISA";
```



strdup()函数可以把字符串复制到堆上：

```
char *copy = strdup(s);
```

1. **strdup()函数计算出字符串的长度，然后调用malloc()函数在堆上分配相应的空间。**



2. **然后strdup()函数把所有字符复制到堆上的新空间。**

也就是说，`strdup()`总是在堆上创建空间，而不是在栈上，因为栈用来保存局部变量，而局部变量很快就会被清除。

因为`strdup()`把新字符串放在堆上，所以千万记得要用**free()**函数释放空间。

用strdup()修复代码

可以用strdup()修复原来的create()函数：

```
island* create(char *name)
{
  island *i = malloc(sizeof(island));
  i->name = strdup(name);
  i->opens = "09:00";
  i->closes = "17:00";
  i->next = NULL;
  return i;
}
```

你会发现我们只需要对name字段使用strdup()函数就行了，知道为什么吗？

因为我们把open和close字段设为了字符串字面值。还记得存储器的分布图吗？字符串字面值位于存储器的**只读**区域，该区域专门用来分配**常量**。把open和close的值设为常量，即使不复制也无所谓，因为它们不会改变。但为了防止出错必须复制name数组，因为后面的代码可能修改它。

这里没有蠢问题

问： **如果island结构用数组保存岛名，而不是字符指针，还需要用strdup()吗？**

答： 不需要，如果用数组，每个island结构都会保存自己的副本，不需要你自己创建。

问： **那为什么要在数据结构中使用字符指针而不是字符数组呢？**

答： 字符指针不会限制字符串的大小。如果用字符数组，需要提前决定字符串的长度。

能改好吗？

为了验证这些修改能否修复代码，我们再次运行原来的代码：

```
File Edit Window Help CoconutAirways
> ./test_flight
Atlantis
Titchmarsh Island
Name: Atlantis
 open: 09:00-17:00
Name: Titchmarsh Island
 open: 09:00-17:00
```

现在，代码正确工作了。用户每输入一个岛名，create()函数都把它保存在了新的字符串中。

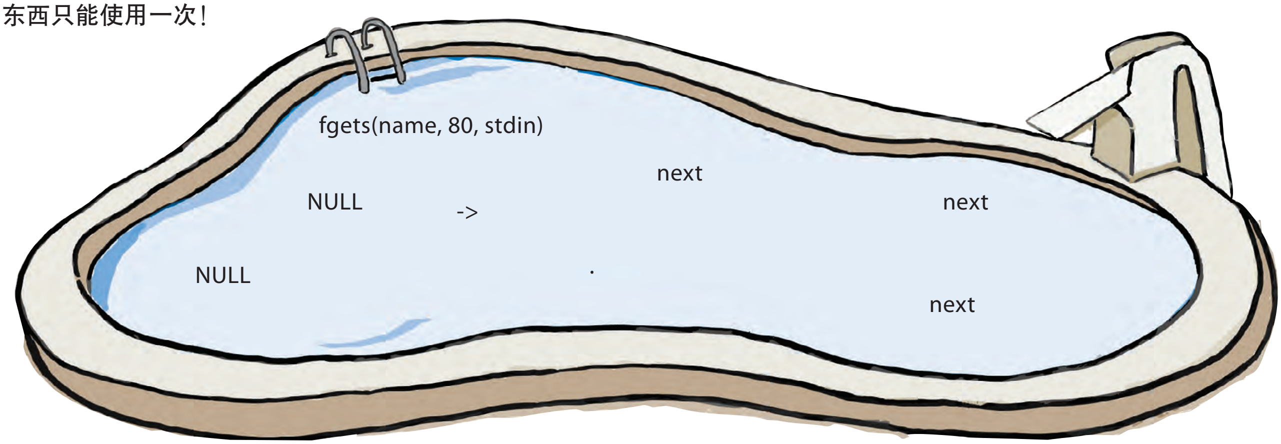
好了，创建岛屿数据的函数你已经有了，我们用它来根据文件创建链表。

游泳池拼图

真倒霉！用来创建飞行之旅的代码掉进了游泳池！你的**任务**是从泳池中取出代码片段，填入以下的空白横线处。你的**目标**是重构程序，程序从标准输入读取一列岛名，然后将它们连成链表。每个代码片段只能使用一次，但不是所有片段都用得到。

```
island *start = NULL;
island *i = NULL;
island *next = NULL;
char name[80];
for(; ............ != ............; i = ............) {
  next = create(name);
  if (start == NULL)
    start = ............;
  if (i != NULL)
    i ............ ............ = next;
}
display(start);
```

注意：游泳池中的每样东西只能使用一次！



游泳池拼图解答

真倒霉！用来创建飞行之旅的代码掉进了游泳池！你的**任务**是从泳池中取出代码片段，填入以下空白横线处。你的**目标**是重构程序，程序从标准输入读取一列岛名，然后将它们连成链表。

```
island *start = NULL;
island *i = NULL;
island *next = NULL;
char name[80];
for(; fgets(name, 80, stdin) != NULL ; i = next ) {
  next = create(name);
  if (start == NULL)
    start = next ;
  if (i != NULL)
    i -> next = next;
}
display(start);
```

从标准输入读取字符串。

在每次循环的最后，将i设为下一座我们要创建的岛屿。

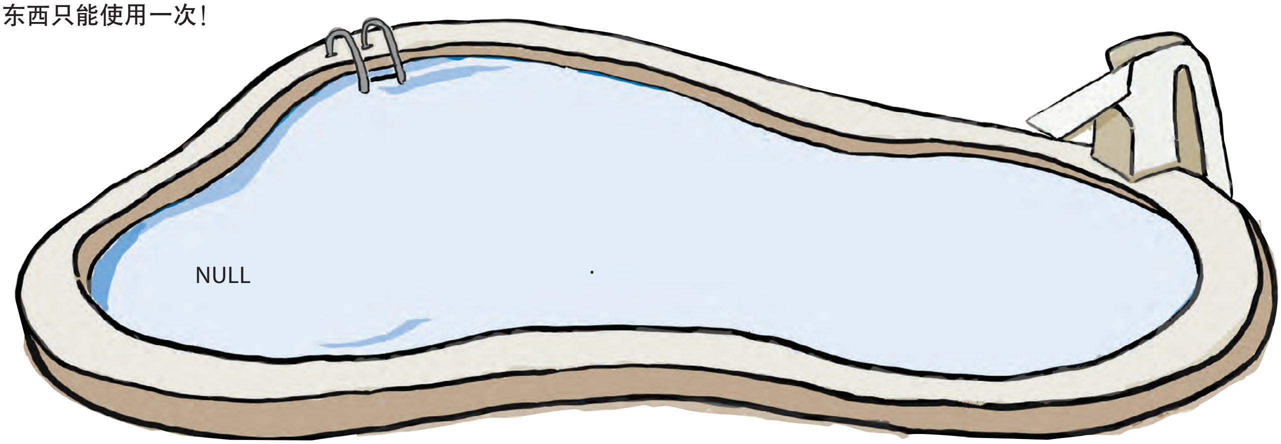
创建岛屿。

一直循环，直到用户不想再输入字符串。

第一遍循环时，start的值是NULL，因此将start设为第一座岛。

别忘了，i是一个指针，我们将使用“->”表示法。

注意：游泳池中的每样东西只能使用一次！



磨笔上阵

等等！还没完呢。别忘了，只要用malloc()函数分配了空间，就需要用free()函数释放它们。到目前为止，程序用malloc()在堆上创建了island链表，但当用完时没有释放这些空间，下面来写这部分代码。

以下是release()函数的开头部分，只要把第一座岛的指针传给它，release()就会释放链表使用的所有存储器：

```
void release(island *start)
{
  island *i = start;
  island *next = NULL;
  for (; i != NULL; i = next) {
    next = ......................;
     ......................;
     ......................;
  }
}
```

好好想想，释放存储器时，要释放哪些东西？只有island，或者还有其他东西？应该按什么顺序释放它们？

磨笔上阵解答

等等！还没完呢。别忘了，只要用malloc()函数分配了空间，就需要用free()函数释放它们。到目前为止，程序用malloc()在堆上创建了island链表，但当用完时没有释放这些空间，下面来写这部分代码。

以下是release()函数的开头部分，只要把第一座岛的指针传给它，release()就会释放链表使用的所有存储器：

```
void release(island *start)
{
  island *i = start;
  island *next = NULL;
  for (; i != NULL; i = next) {
    next = i->next;
    free(i->name);
    free(i);
  }
}
```

把next设为指向下一座岛的指针。

首先，需要释放用strdup()创建的name字符串。

只有先释放name，才能释放island结构。

如果先释放了island结构，就再也找不到name去释放了。

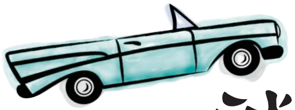
释放存储器时，要释放哪些东西？只有island，或者还有其他东西？应该按什么顺序释放它们？

用完后释放存储器

既然有释放链表的函数，就需要在用完链表以后调用它。程序只需要显示链表的内容，显示完以后就可以释放它：

```
display(start);
release(start);
```

写完后你可以测试代码。



试驾

编译代码，运行程序并把文件传给它，看看发生了什么。

```
File Edit Window Help FreeSpaceYouDon'tNeed
> ./tour < trip1.txt
Name: Delfino Isle
 Open: 09:00-17:00
Name: Angel Island
 Open: 09:00-17:00
Name: Wild Cat Island
 Open: 09:00-17:00
Name: Neri's Island
 Open: 09:00-17:00
Name: Great Todday
 Open: 09:00-17:00
Name: Ramita de la Baya
 Open: 09:00-17:00
Name: Island of the Blue Dolphins
 Open: 09:00-17:00
Name: Fantasy Island
 Open: 09:00-17:00
Name: Farne
 Open: 09:00-17:00
Name: Isla de Muert
 Open: 09:00-17:00
Name: Tabor Island
 Open: 09:00-17:00
Name: Haunted Isle
 Open: 09:00-17:00
Name: Sheena Island
 Open: 09:00-17:00
```

程序正确运行了。记住，你无法知道文件有多大。在这个例子中，即使你不把所有数据都保存在存储器中，也能把它们打印出来，但只有把数据放进存储器，才能自由地处理它们。你可以在旅途中添加或删除一些站点，还可以重新调整旅行的顺序，或扩充它。

有了动态分配存储器，就能在运行时创建需要的存储器。使用malloc()与free()，可以访问动态堆存储器。

今夜话题：栈与堆在讨论他们之间的差异

栈： 堆？你在家吗？

堆： 平时这个时候很少见到你，最近忙啥呢？

栈： 刚刚从一个函数返回，实在不好意思，最近一直在整理东西……

堆： 你都做了些什么？

栈： 代码刚刚退出函数，我需要释放局部变量的空间。

堆： 你应该活得更简单一点，放轻松……

栈： 也许你是对的，我能坐下么？

堆： 要啤酒吗？不用管这个帽子，扔一边就行了。

栈： 这个东西是你的吗？

堆： 嘿，你找到了披萨！太好了，我找了它一个礼拜。

栈： 你应该让别人来帮忙收拾一下这里。

堆： 不用担心，在线点餐程序把披萨留在了这里，他应该还会回来的。

栈： 你怎么知道？万一他忘了呢？

堆： 他重新联络过我，他调用了`free()`。

栈： 嗯？你确定？这个程序是写“打兔子”游戏的那个家伙写的么？存储器泄漏得到处都是，满地的兔子结构，我都没法走路了。到处都是垃圾，太可怕了。

栈：

你这样很不负责任。

瞎折腾？我从来不瞎折腾！你可能想要一张纸巾……

你应该更合理地维护存储器。

你真邋遢。

为什么你不做“垃圾收集”？！

一丁点儿整理工作而已，现在你什么也不干！！！

（哭）对不起，我受不了了，这里实在是太乱了。

（擤鼻涕）谢谢，等一下，这是什么？

堆：

喂，清理存储器可不是我的责任。有人申请空间，我就给他，我会把空间留在那里，直到他叫我清理它。

也许吧，但使用起来很简单，不像你那么瞎折腾。

（打嗝）什么？我只是说你很难追踪。

随便，我一向宽以待人。如果程序想把存储器搞得乱七八糟，那也不是我的责任。

是随遇而安。

又来了……

放松。

嘿，你的鼻子“溢出”了，用这个……

“打兔子”游戏的排行榜。别担心，我想程序不需要再用它了。

这里没有蠢问题

问：为什么“堆”要叫做“堆”？

答：因为计算机不会自动组织它，它只是一大“堆”数据而已。

问：什么是“垃圾收集（garbage colloection）”？

答：一些语言会跟踪你在堆上分配的数据，当你不再使用这些数据时，就会释放它们。

问：为什么C语言没有“垃圾收集”？

答：C语言非常古老，发明它的时候，绝大多数语言都没有自动“垃圾回收”机制。

问：这个例子中，复制island名字的原因我知道，但为什么open和close的值不需要复制？

答：open和close的值都设为了字符串字面值，而字符串字面值无法更新，即便多项数据引用了相同字符串也没关系。

问：strdup()函数实际上会调用malloc()函数吗？

答：这取决于C标准库是如何实现的，不过通常情况下，是这样的。

问：我需要在程序结束前释放所有数据吗？

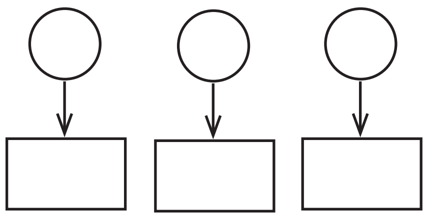
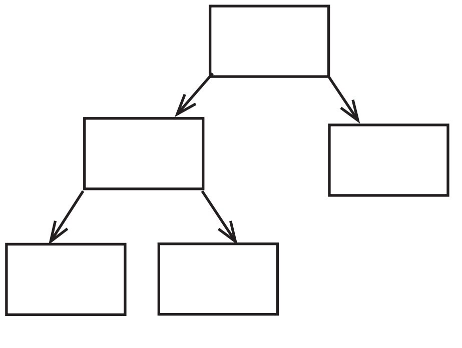
答：不必，操作系统会在程序结束时清除所有存储器。不过，你还是应该显式释放你创建的每样东西，这是一种好的习惯。

要点

- 可以用动态数据结构保存可变数量的数据项。
- 可以很方便地在链表这种数据结构中插入数据项。
- 在C语言中，动态数据结构通常用递归结构来定义。
- 递归结构中有一个或多个指向相同结构的指针。
- 栈用来保存局部变量，它由计算机管理。
- 堆用来保存长期使用的数据，可以用`malloc()`分配堆空间。
- `sizeof`运算符告诉你一个结构需要多少空间。
- 数据会一直留在堆上，直到用`free()`释放它。

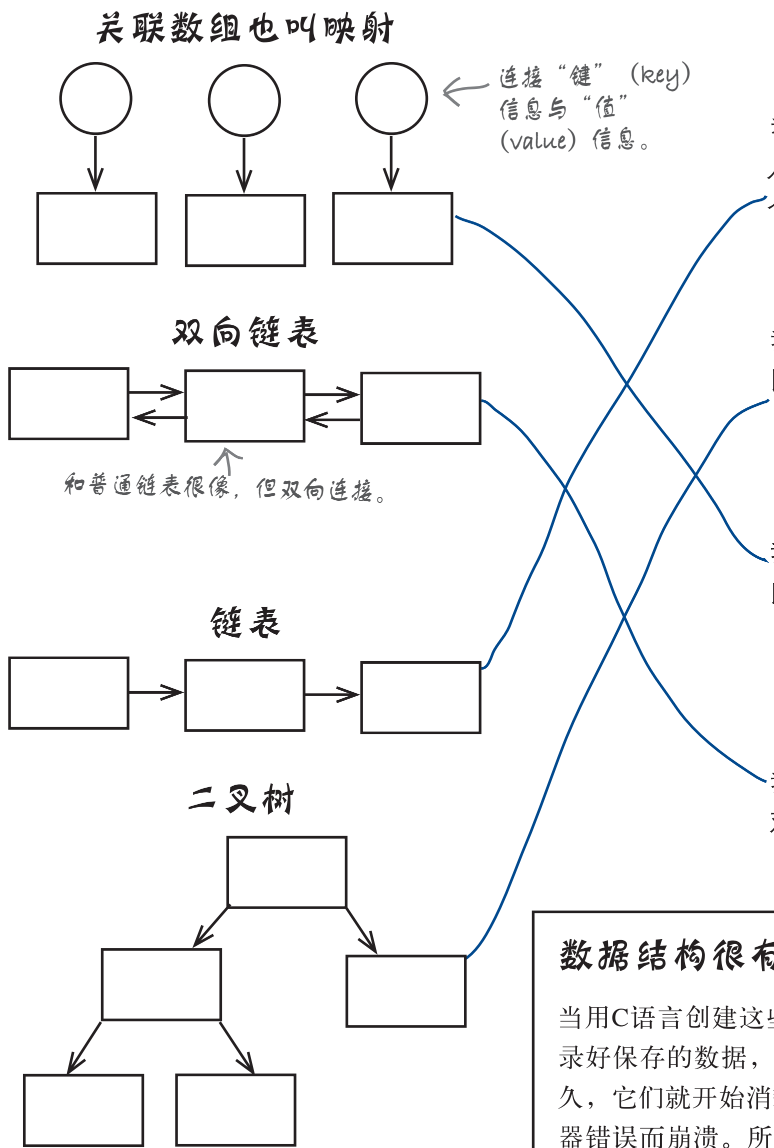
猜猜我是谁？

你已经学会了用C语言创建链表，但链表并不是唯一的数据结构，可能还需要创建其他数据结构。下面是一些其他数据结构的例子，看你能不能把数据结构与使用说明连起来。

数据结构	说明
	我可以用来保存一串数据项，并使插入新数据项变得简单，但只能沿着一个方向处理我。
	我的每一项都与其他两项相连，我可以用来保存层次信息。
	我可以用来连接两种不同类型的数据，比如可以用我连接人名与电话号码。
	我的每一项都与其他两项相连，可以双向处理我。

猜猜我是谁？解答

你已经学会了用C语言创建链表，但链表并不是唯一的数据结构，可能还需要创建其他数据结构。下面是一些其他数据结构的例子，你将把数据结构与使用说明连起来。



说明

我可以用来保存一串数据项，并使插入新数据项变得简单，但只能沿着一个方向处理我。

我的每一项都与其他两项相连，我可以用来保存层次信息。

我可以用来连接两种不同类型的数据，比如可以用我连接人名与电话号码。

我的每一项都与其他两项相连，可以双向处理我。

数据结构很有用，但要小心使用！

当用C语言创建这些数据结构时需要非常小心，如果没有记录好保存的数据，就很可能把不用的数据留在堆上。时间一久，它们就开始消耗机器上的存储器，程序也可能因为存储器错误而崩溃。所以，你必须学会如何追查代码中的存储器泄漏，并学会如何修复它们……

最高机密

华盛顿特区美国司法部联邦调查局

发件人：小埃德加 · 胡佛（局长）

主题：政府专家系统中的可疑泄漏

麻省剑桥市分局报告“可疑人物识别专家系统”（Suspicious Persons Identification Expert System，SPIES）中存在可疑泄漏。据可靠消息和几位精通软件的线人透露，该泄漏是由“临时工”编码造成的，涉案人员未知。

一名曾提供过可靠消息并声称与当事人有着密切关系的线人表示，该漏洞是由存储器中某块区域数据的管理不善所引起的，黑客兄弟会的人把该区域称为“堆”。

现在，我赋予你查看专家系统源代码以及动用FBI软件工程实验室所有资源的权力，请考虑种种迹象，仔细地分析案件的每一个细节，找到并修复这个漏洞。

只许成功不许失败。

此致

敬礼！

物证一：源代码

以下是“可疑人物识别专家系统”（SPIES）的源代码。这款软件可以用来记录嫌疑犯，并辨认他们。你不需要现在就细读代码，但请留一份副本，调查的过程中可能会用到它。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

typedef struct node {
  char *question;
  struct node  *no;
  struct node  *yes;
} node;

int yes_no(char *question)
{
  char answer[3];
  printf("%s? (y/n): ", question);
  fgets(answer, 3, stdin);
  return answer[0] == 'y';
}

node* create(char *question)
{
  node *n = malloc(sizeof(node));
  n->question = strdup(question);
  n->no = NULL;
  n->yes = NULL;
  return n;
}

void release(node *n)
{
  if (n) {
    if (n->no)
      release(n->no);
    if (n->yes)
      release(n->yes);
    if (n->question)
      free(n->question);
    free(n);
  }
}
```

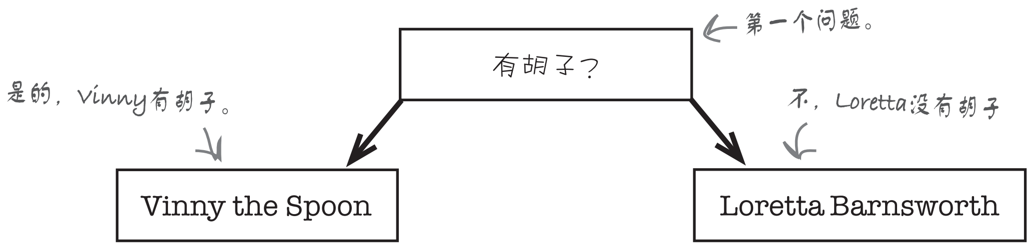
```
int main()
{
  char question[80];
  char suspect[20];
  node *start_node = create("Does suspect have a mustache");
  start_node->no = create("Loretta Barnsworth");
  start_node->yes = create("Vinny the Spoon");

  node *current;
  do {
    current = start_node;
    while (1) {
      if (yes_no(current->question))
      {
        if (current->yes) {
          current = current->yes;
        } else {
          printf("SUSPECT IDENTIFIED\n");
          break;
        }
      } else if (current->no) {
        current = current->no;
      } else {

         /* Make the yes-node the new suspect name */
         printf("Who's the suspect? ");
         fgets(suspect, 20, stdin);
         node *yes_node = create(suspect);
         current->yes = yes_node;

         /* Make the no-node a copy of this question */
         node *no_node = create(current->question);
         current->no = no_node;

         /* Then replace this question with the new question */
         printf("Give me a question that is TRUE for %s but not for %s? ", suspect,
            current->question);
         fgets(question, 80, stdin);
         current->question = strdup(question);

         break;
      }
    }
  } while(yes_no("Run again"));
  release(start_node);
  return 0;
}
```

SPIES系统综述

SPIES程序是一个专家系统，通过学习嫌疑犯的特征，它能够辨认他们。你往系统中输入的人越多，软件也就学得越多，也就越聪明。

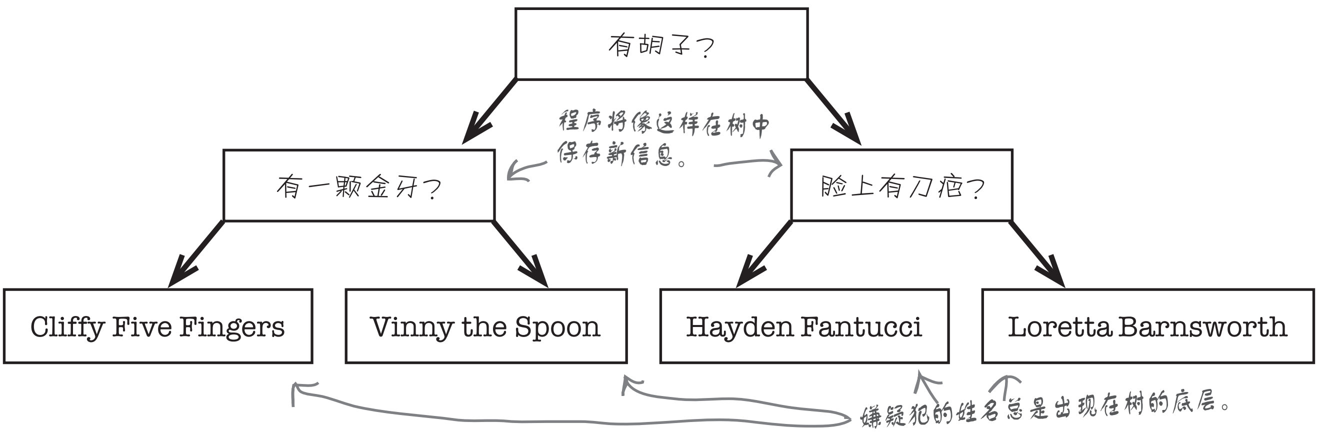
程序建立一棵嫌疑犯树

程序用一棵二叉树记录数据。**二叉树**可以把一条数据与另外两条数据连接在一起，方法如下：

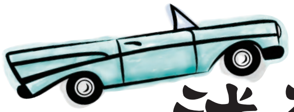


这是程序启动时的数据组成。二叉树中第一项（也叫**节点**）保存了一个问题：“嫌疑犯有胡子吗？”它与另外两个节点相连：一个代表yes，另一个代表*no*，*yes*与*no*节点中分别保存了一个嫌疑犯的名字。

为了辨认嫌疑犯，程序将利用这棵树来向用户提出一系列问题。如果找不到嫌疑犯，程序会要求用户输入这名新嫌疑犯的名字，和一些用于辨认他的信息。程序会把这些信息保存在二叉树中，随着程序学到的东西越来越多，二叉树会渐渐长高。



运行程序看看。



试驾

特工编译了SPIES程序，并进行了测试，结果如下：

```
File  Edit  Window  Help  TrustNoone
> gcc spies.c -o spies && ./spies
Does suspect have a mustache? (y/n): n
Loretta Barnsworth? (y/n): n
Who's the suspect? Hayden Fantucci
Give me a question that is TRUE for Hayden Fantucci
 but not for Loretta Barnsworth? Has a facial scar
Run again? (y/n): y
Does suspect have a mustache? (y/n): n
Has a facial scar
? (y/n): y
Hayden Fantucci
? (y/n): y
SUSPECT IDENTIFIED
Run again? (y/n): n
>
```

第一遍运行时，程序未能辨认出嫌疑犯Hayden Fantucci。但当用户输入了嫌疑犯的详细信息，程序便获取了足够的信息，第二遍运行时就能辨认出Fantucci了。

真聪明。有问题吗？

有人在实验室中连续用了这个系统几个小时，他注意到，尽管程序看起来运行正确，但却多用了一倍存储器。

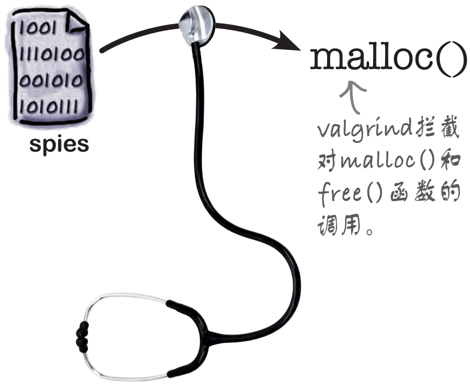
所以我们把**你**请来。源代码深处藏着一段代码，它在堆上分配存储器，但从不释放。你可以搬个椅子坐下来，然后通读所有代码，并祈祷能发现问题所在。通常情况下，很难发现存储器泄漏。

也许你应该跑一趟软件实验室……

软件取证：使用valgrind

在一个像SPIES这样庞大而复杂的程序里面，查找错误十分费时，所以C黑客写了一些帮助查错的工具。其中，有一个叫**valgrind**的工具，它用于Linux操作系统中。

valgrind通过伪造**malloc()**可以监控分配在堆上的数据。当程序想分配堆存储器时，valgrind将会拦截你对malloc()和free()的调用，然后运行自己的malloc()和free()。valgrind的malloc()会记录调用它的是哪段代码和分配了哪段存储器。程序结束时，valgrind会汇报堆上有哪些数据，并告诉你这些数据是由哪段代码创建的。



准备好代码：添加调试信息

在使用valgrind运行代码前，你不需要做任何修改，甚至不需要重新编译代码。但为了发挥valgrind的最大威力，应当在可执行文件中包含调试信息。**调试**信息是编译时打包到可执行文件中的附加数据，比如某段代码在源文件中的行号。只要有调试信息，valgrind就能提供更多有助于发现存储器泄漏的信息。

为了在可执行文件中加入调试信息，需要加上-g开关，并重新编译源代码。

```
gcc -g spies.c -o spies
```

-g开关告诉编译器要记录要编译代码的行号。

真相只有一个：审问代码

为了弄明白valgrind是如何工作的，我们在Linux命令行中打开它，用它审问几次SPIES程序。

可以在http://valgrind.org查看valgrind是否支持你的操作系统，并查看如何安装。

首先，我们用程序来辨认默认嫌疑犯Vinny the Spoon。在命令行启动valgrind，加上--leak-check=full选项，并把你想运行的程序传给valgrind：

```
File Edit Window Help valgrindRules
> valgrind --leak-check=full ./spies
==1754== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
Does suspect have a mustache? (y/n): y
Vinny the Spoon? (y/n): y
SUSPECT IDENTIFIED
Run again? (y/n): n
==1754== All heap blocks were freed -- no leaks are possible
```

反复使用valgrind，收集更多证据

当SPIES程序退出时，堆上什么都没有。再次运行程序，教程序辨认一个叫Hayden Fantucci的新嫌疑犯，看看会发生什么。

```
File  Edit  Window  Help  valgrindRules
> valgrind --leak-check=full ./spies
==2750== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
Does suspect have a mustache? (y/n): n
Loretta Barnsworth? (y/n): n
Who's the suspect? Hayden Fantucci
Give me a question that is TRUE for Hayden Fantucci
 but not for Loretta Barnsworth? Has a facial scar
Run again? (y/n): n
==2750== HEAP SUMMARY:
==2750==     in use at exit: 19 bytes in 1 blocks
==2750==   total heap usage: 11 allocs, 10 frees, 154 bytes allocated
==2750== 19 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2750==    at 0x4026864: malloc (vg_replace_malloc.c:236)
==2750==    by 0x40B3A9F: strdup (strdup.c:43)
==2750==    by 0x8048587: create (spies.c:22)
==2750==    by 0x804863D: main (spies.c:46)
==2750== LEAK SUMMARY:
==2750==    definitely lost: 19 bytes in 1 blocks
>
```

19字节的数据留在了堆上。

分配了11次存储器，但只释放了10次。

你能从这几行中看出什么吗？

为什么是19字节？你能推测出什么吗？

这次valgrind发现了存储器泄漏

程序结束时，似乎有19字节的信息留在了堆上，valgrind告诉你以下几件事：

- **分配了19字节的存储器，但没有释放。**
- **看起来我们分配了11次存储器，但只释放了10次。**
- **你能从这几行中看出什么吗？**
- **为什么是19字节？你能推测出什么吗？**

有很多信息，下面我们来分析它们。

推敲证据

好了，既然你已经用valgrind收集了不少证据，下面就来分析这些证据，看看能否得出什么结论。

1. 定位

运行了两次代码，第一次没有任何问题。只有当输入一个新嫌疑犯的名字时，存储器才会泄漏。这条线索十分重要，因为它说明泄漏不可能发生在第一次运行的代码中。回过去看源代码，问题应该发生在以下代码中：

```
} else if (current->no) {
  current = current->no;
} else {

   /* Make the yes-node the new suspect name */
   printf("Who's the suspect? ");
   fgets(suspect, 20, stdin);
   node *yes_node = create(suspect);
   current->yes = yes_node;

   /* Make the no-node a copy of this question */
   node *no_node = create(current->question);
   current->no = no_node;

   /* Then replace this question with the new question */
   printf("Give me a question that is TRUE for %s but not for %s? ",
           suspect, current->question);
   fgets(question, 80, stdin);
   current->question = strdup(question);

   break;
}
```

2. valgrind提供的线索

当用valgrind运行代码并添加一名嫌疑犯时，程序分配了11次存储器，但只释放了10次，这说明什么？

valgrind告诉你程序结束时有19个字节的数据留在了堆上。看一下源代码，哪条数据像是有19字节？

最后，下面这段valgrind的输出告诉你什么？

```
==2750== 19 bytes in 1 blocks are definitely lost in loss record 1 of 1
==2750==    at 0x4026864: malloc (vg_replace_malloc.c:236)
==2750==    by 0x40B3A9F: strdup (strdup.c:43)
==2750==    by 0x8048587: create (spies.c:22)
==2750==    by 0x804863D: main (spies.c:46)
```

提问

仔细考虑这些证据，然后回答以下问题。

1. 有几条数据留在了堆上？

2. 哪条数据留在了堆上？

3. 哪一行或哪几行代码导致了泄漏？

4. 如何修复泄漏？

提问

你仔细考虑了这些证据，并回答了以下问题。

1. 有几条数据留在了堆上？

有一条数据。

2. 哪条数据留在了堆上？

字符串“Loretta Barnsworth”，18个字符外加一个字符串终结符。

3. 哪一行或哪几行代码导致了泄漏？

create()函数本身不会导致泄漏，因为第一遍运行的时候没有发生，

所以一定是下面这行strdup()出了问题：

```
current->question = strdup(question);
```

4. 如何修复泄漏？

current->question已经指向了堆上的某个question，因此在分配新的question之前要先释放它：

```
free(current->question);
current->question = strdup(question);
```

最终审判

既然修改了代码，再用valgrind运行一次：

```
File  Edit  Window  Help  valgrindRules
> valgrind --leak-check=full ./spies
==1800== Copyright (C) 2002-2010, and GNU GPL'd, by Julian Seward et al.
Does suspect have a mustache? (y/n): n
Loretta Barnsworth? (y/n): n
Who's the suspect? Hayden Fantucci
Give me a question that is TRUE for Hayden Fantucci
 but not for Loretta Barnsworth? Has a facial scar
Run again? (y/n): n
==1800== All heap blocks were freed -- no leaks are possible
>
```

泄漏已修复

你运行了和刚刚一样的测试数据，但这次程序清理了堆上的所有东西。

你破案了吗？即使这次没能发现泄漏并修复它也不用担心，存储器泄漏是C程序中最难发现的错误。事实上，现在你用的很多C程序都隐藏着一些存储器错误，所以valgrind工具就非常有用。

- 发现泄漏。
- 定位泄漏。
- 检验泄漏是否修复。

这里没有蠢问题

问：valgrind说泄漏的存储器是在第46行创建的，但我们却修改了另一行代码，为什么？

答： 虽然数据“Loretta...”是由第46行代码放到堆上的，但泄漏却发生在变量（current->question）重新赋值的那一刻，因为当时变量指向的“Loretta...”还没有释放。创建数据不会发生泄漏，只有当程序失去了所有对数据的引用才会导致泄漏。

问：我的Mac/Windows/FreeBSD系统可以安装valgrind吗？

答： 在http://valgrind.org上可以查看valgrind最新发行版的详细信息。

问：那么valgrind是怎么拦截malloc()与free()的？

答： malloc()和free()包含在C标准库中，而valgrind有一个库，里面有它自己的malloc()与free()。当用valgrind运行程序时，程序会使用valgrind的函数，而不是C标准库中的函数。

问：为什么编译器在编译代码时不默认包含调试信息？

答： 因为调试信息会使可执行文件变大，同时也可能让程序变得更慢。

问：valgrind这个名字的由来是什么？

答： valgrind是英灵殿[①]入口的名字，而valgrind（程序）为你打开了一扇通向计算机堆的大门。

要点

- valgrind可以检查存储器泄漏。
- valgrind通过拦截对malloc()与free()的调用来工作。
- 程序在停止运行时，valgrind会打印留在堆上数据的详细信息。
- 编译代码时，如果在可执行文件中加上调试信息，valgrind可以提供更多信息。
- 多次运行程序可以缩小泄漏的范围。
- valgrind可以告诉你源文件的哪行代码把数据放到了堆上。
- valgrind可以用来检验泄漏是否已修复。

① 北欧神话中，死亡之神奥丁用来款待阵亡将士英灵的殿堂。——译者注

C语言工具箱

你已经学完了第6章，现在你的工具箱又加入了数据结构与动态存储。关于本书的提示工具条的完整列表，请见附录ii。

- 链表比数组更容易扩展。
- 在链表中插入数据很方便。
- 链表是动态数据结构。
- 动态数据结构使用递归结构。
- 递归结构包含一个或多个链向相同结构数据的链接。
- malloc()在堆上分配存储器。
- free()释放堆上的存储器。
- 与栈不同，堆存储器不会自动释放。
- 栈用来保存局部变量。
- strdup()会把字符串复制到堆上。
- 存储器泄漏是指存储器分配出去以后，你再也访问不到。
- valgrind可以帮助追踪存储器泄漏。

7 高级函数

发挥函数的极限

基本函数很好用，但有时需要更多功能。

到目前为止，你只关注了一些基本的东西，为了达成目标，需要更多的功能与灵活性。本章你将学习如何把函数作为参数传递，从而提高代码的智商，并学会用比较器函数排序，最后还将学会使用可变参数函数让代码伸缩自如。

寻找真命天子……

到目前为止，你已经用过了书中很多C函数，事实上还有很多方法可以让它们变得更强大，只要学会正确使用这些方法，就可以用更少的代码做更多的事。

怎么做？我们来看一个例子。假设你想过滤某个字符串数组中的数据，只显示其中部分字符串：

```
int NUM_ADS = 7;
char *ADS[] = {
  "William: SBM GSOH likes sports, TV, dining",
  "Matt: SWM NS likes art, movies, theater",
  "Luis: SLM ND likes books, theater, art",
  "Mike: DWM DS likes trucks, sports and bieber",
  "Peter: SAM likes chess, working out and art",
  "Josh: SJM likes sports, movies and theater",
  "Jed: DBM likes theater, books and dining"
};
```

下面来写代码，用字符串函数过滤数组。

代码冰箱贴

请完成find()函数，用它过滤出列表中所有运动迷，同时他们不能是Bieber的粉丝。

注意：有的代码片段可能不会用到。

```
void find()
{
  int i;
  puts("Search results:");
  puts("------------------------------------");

  for (i = 0; i .................. .................. ; i++) {

    if (.................. (.................. , ..................)

        .................. .................. ..................(.................. , ..................)) {

      printf("%s\n", ADS[i]);
    }
  }
  puts("------------------------------------");
}
```

strcmp

strstr　ADS[i]　ADS[i]　"sports"

NUM_ADS

<　strstr　!　&&　||　"bieber"　strcmp

代码冰箱贴解答

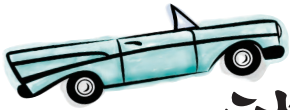
请完成`find()`函数，用它过滤出列表中所有运动迷，同时他们不能是Bieber的粉丝。

```
void find()
{
  int i;
  puts("Search results:");
  puts("------------------------------------");

  for (i = 0; i < NUM_ADS ; i++) {
    if ( strstr ( ADS[i] , "sports" )
        && ! strstr ( ADS[i] , "bieber" )) {
      printf("%s\n", ADS[i]);
    }
  }
  puts("------------------------------------");
}
```

strcmp

||

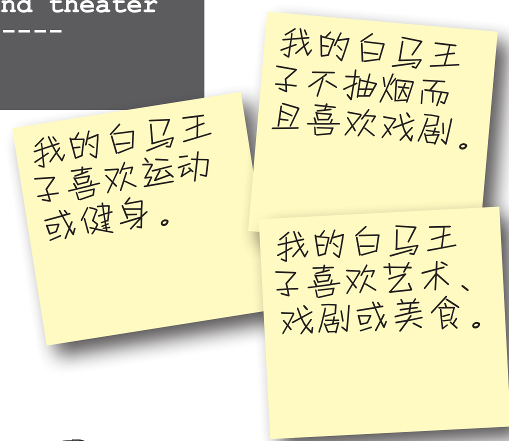
strcmp



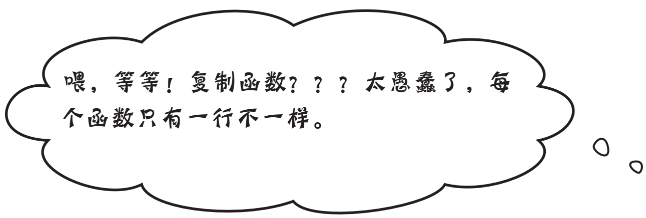
试驾

假如把函数和数据都放在一个叫find.c的文件中，就可以像这样编译并运行程序：

```
File  Edit  Window  Help  FindersKeepers
> gcc find.c -o find && ./find
Search results:
------------------------------------
William: SBM GSOH likes sports, TV, dining
Josh: SJM likes sports, movies and theater
------------------------------------
>
```



和预期一样，find()函数循环遍历了数组，然后找到了匹配的字符串。既然有了基本代码，就不难复制出搜索其他内容的函数。



没错，复制函数会产生很多重复代码。

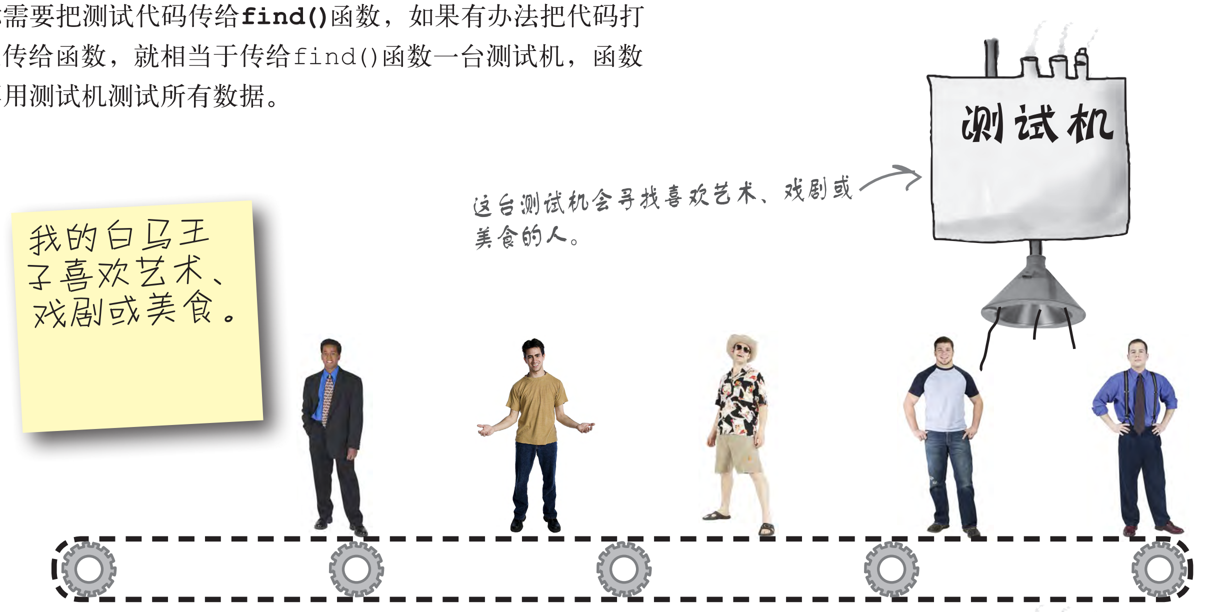
C程序经常会执行一些大同小异的任务，现在find()函数为了搜索匹配字符串，会遍历数组中所有元素，并测试每个字符串，而这些测试会写死在代码中，也就是说函数永远只能做一种测试。

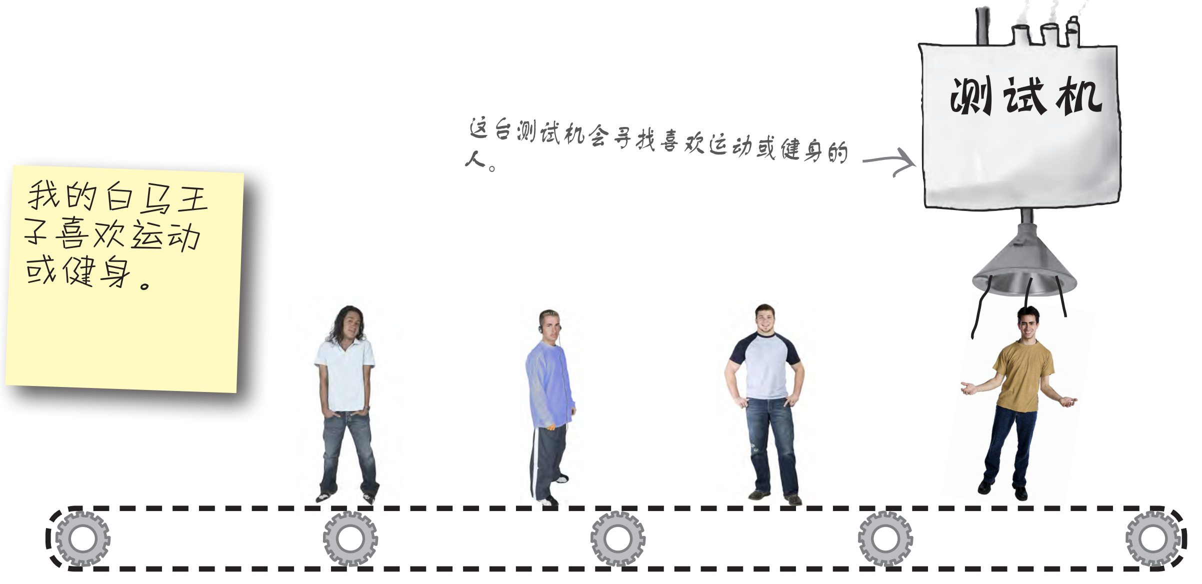
当然也可以把字符串作为参数传递给函数，让函数搜索不同的子串，但这样find()还是无法检查3个字符串，比如“arts”、“theater”和“dining”。你需要的是一种截然不同的技术。

你需要更高端的东西……

把代码传给函数

你需要把测试代码传给**find()**函数，如果有办法把代码打包传给函数，就相当于传给find()函数一台测试机，函数再用测试机测试所有数据。





这样一来find()函数中大部分代码可以原封不动。代码还是要检查数组中所有元素，并显示相同的输出，只是测试数组元素的代码是你传给它的。

把函数名告诉find()

假设你从原来的代码中提取出了搜索条件，并把它改写成函数：

```
int sports_no_bieber(char *s)
{
  return strstr(s, "sports") && !strstr(s, "bieber");
}
```

现在，只要有办法把函数名作为参数传给find()，就能在find()中注入测试了。

```
void find( function-name match)
{
  int i;
  puts("Search results:");
  puts("--------------------------------------");
  for (i = 0; i < NUM_ADS; i++) {
    if ( call-the-match-function (ADS[i])) {
      printf("%s\n", ADS[i]);
    }
  }
  puts("--------------------------------------");
}
```

match指定了含测试代码函数的名称。

这个地方，你需要以某种方式调用函数，函数名由参数match给出。

只要能找到一种把函数名传给find()的方法，以后就可以做任何类型的测试了。只要能写一个接收字符串并返回真或假的函数，就可以复用同一个find()函数。

```
find(sports_no_bieber);
find(sports_or_workout);
find(ns_theater);
find(arts_theater_or_dining);
```

如何在形参中保存函数名？如果你有函数名，又如何用它来调用函数呢？

函数名是指向函数的指针①……

可能你已经猜到了，这一定和指针有关。想想函数名到底是什么，它可以引用某段代码。这就是指针：引用存储器中某样东西的方法。

在C语言中，函数名也是指针变量。当你创建了一个叫go_to_warp_speed(int speed)函数的同时也会创建了一个叫go_to_warp_speed的指针变量，变量中保存了函数的地址。只要把函数指针类型的参数传给find()，就能调用它指向的函数了。

```
int go_to_warp_speed(int speed)
{
  dilithium_crystals(ENGAGE);
  warp = speed;
  reactor_core(c, 125000 * speed, PI);
  clutch(ENGAGE);
  brake(DISENGAGE);
  return 0;
}
```

创建函数的同时也创建了一个同名函数指针。

指针中保存了函数的地址。

```
go_to_warp_speed(4);
```

当调用函数时，你在使用函数指针。

下面来看看函数指针的语法。

① 两者并不完全相同，函数名是L-value，而指针变量是R-value，因此函数名不能像指针变量那样自加或自减。——译者注

……没有函数类型

在C语言中声明指针很容易，假设你要声明int类型的指针，只需要在类型名后加一个星号，即int *。但C语言中没有函数类型，所以不能用function *声明函数指针。

```
int *a;
```

声明int指针……

```
function *f;
```

……但不能这样声明函数指针。

为什么C语言没有函数类型

C语言没有函数类型，因为函数的类型不止一种。当你创建函数时，可以改变很多东西，例如返回类型或形参列表，函数的类型是由这些东西的组合定义的。

```
int go_to_warp_speed(int speed)
{
   ...
}

char** album_names(char *artist, int year)
{
   ...
}
```

函数有不同的返回类型和形参，所以它有许多不同的类型。

因此函数指针的表示方法更复杂……

如何创建函数指针

假如想要创建指针变量，用来保存上一页中函数的地址，必须像这样做：

```
int (*warp_fn)(int);
warp_fn = go_to_warp_speed;
warp_fn(4);
```

创建一个叫warp_fn的变量，用来保存go_to_warp_speed()函数的地址。

相当于调用go_to_warp_speed(4)。

```
char** (*names_fn)(char*,int);
names_fn = album_names;
char** results = names_fn("Sacha Distel", 1972);
```

创建一个叫names_fn的变量，用来保存album_names()函数的地址。

看起来很复杂，对吗？

但就得那么复杂，因为需要把函数的返回类型和接收参数类型告诉C编译器。一旦声明了函数指针变量，就可以像其他变量一样使用它，可以对它赋值，也可以把它加到数组中，还可以把它传给函数……

……回去看看find()的代码……

这里没有蠢问题

问： char**是什么意思？是不是打错了？

答： char**是一个指针，通常用来指向字符串数组。

看看其他人想要找的男生类型，你能否为每类搜索创建函数？第一个已经写好。

喜欢运动但不喜欢Bieber。

```
int sports_no_bieber(char *s)
{
  return strstr(s, "sports") && !strstr(s, "bieber");
}
```

我的白马王子喜欢运动或锻炼身体。

```
int sports_or_workout(char *s)
{
    ......................................................................
}
```

我的白马王子喜欢戏剧，而且不抽烟。

```
int ns_theater(char *s)
{
    ......................................................................
}
```

我的白马王子喜欢艺术、戏剧或美食。

```
int arts_theater_or_dining(char *s)
{
    ......................................................................
}
```

接下来，看看能否完成find()函数：

```
void find( ................................ )
{
  int i;
  puts("Search results:");
  puts("------------------------------------");
  for (i = 0; i < NUM_ADS; i++) {
    if (match(ADS[i])) {
      printf("%s\n", ADS[i]);
    }
  }
  puts("------------------------------------");
}
```

需要传给find()一个叫match的函数指针。

它会调用传进来的match()函数。

你将看到其他人想找什么类型的男生，并为每类搜索创建函数。

喜欢体育运动但不喜欢Bieber。

```
int sports_no_bieber(char *s)
{
  return strstr(s, "sports") && !strstr(s, "bieber");
}
```

我的白马王子喜欢运动或锻炼身体。

```
int sports_or_workout(char *s)
{
   return strstr(s, "sports") || strstr(s, "working out");
}
```

我的白马王子喜欢戏剧，而且不抽烟。

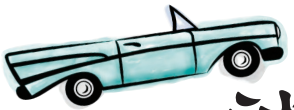
```
int ns_theater(char *s)
{
   return strstr(s, "NS") && strstr(s, "theater");
}
```

我的白马王子喜欢艺术、戏剧或美食。

```
int arts_theater_or_dining(char *s)
{
   return strstr(s, "arts") || strstr(s, "theater") || strstr(s, "dining");
}
```

然后完成find()函数。

```
void find( int (*match)(char*) )
{
  int i;
  puts("Search results:");
  puts("------------------------------------");
  for (i = 0; i < NUM_ADS; i++) {
    if (match(ADS[i])) {
      printf("%s\n", ADS[i]);
    }
  }
  puts("------------------------------------");
}
```



试驾

把你写的这些函数拉出来溜溜，看看它们是怎么工作的。
你需要创建一个程序，依次把函数传给find()：

```
int main()
{
  find(sports_no_bieber);
  find(sports_or_workout);
  find(ns_theater);
  find(arts_theater_or_dining);
  return 0;
}
```

```
File Edit Window Help FindersKeepers
> ./find
Search results:
------------------------------------
William: SBM GSOH likes sports, TV, dining
Josh: SJM likes sports, movies and theater
------------------------------------
Search results:
------------------------------------
William: SBM GSOH likes sports, TV, dining
Mike: DWM DS likes trucks, sports and bieber
Peter: SAM likes chess, working out and art
Josh: SJM likes sports, movies and theater
------------------------------------
Search results:
------------------------------------
Matt: SWM NS likes art, movies, theater
------------------------------------
Search results:
------------------------------------
William: SBM GSOH likes sports, TV, dining
Matt: SWM NS likes art, movies, theater
Luis: SLM ND likes books, theater, art
Josh: SJM likes sports, movies and theater
Jed: DBM likes theater, books and dining
------------------------------------
>
```

这是find(sports_no_bieber)。

这是find(sports_or_workout)。

这是find(ns_theater)。

这是find(arts_theater_or_dining)。

find()函数每次都搜索了不同的内容。有了函数指针，就能把函数传给函数，用更少的代码创建功能更强大的程序，这就是为什么说函数指针是C语言最强大的特性之一。

函数指针狩猎指南

当你身处芦苇丛中，识别函数指针就变得异常困难，下面这个狩猎指南简单易懂，便于携带，而且刚好可以装进C程序员的弹夹包。

返回类型	(*	指针变量	)(	参数类型	)

```
char** (*names_fn)(char*,int)
```

这是你要声明的变量名。

这里没有蠢问题

问： 如果函数指针是指针，为什么调用函数时不需要在它们前面加*？

答： 可加也可不加，可以把代码中的match(ADS[i])换成(*match)(ADS[i])。

问： 我可以用&取得函数的地址吗？

答： 当然，除了find(sports_or_workout)，还可以写find(&sports_or_workout)。

问： 那为什么不这么写？

答： 即使省略*和&，C编译器也能识别它们，这样代码更好读。

用C标准库排序

程序员经常要对数据进行排序。对数字排序很容易，因为数字有大小，但对其他类型来讲，排序可就没那么容易了。

假设现在你面前站着一群人，如何对他们排序呢？按身高？智力？还是魅力？

C标准库的作者在创建排序函数时碰到一个问题：

排序函数如何才能对任何类型的数据进行排序？

用函数指针设置顺序

可能你已经猜到了答案：C标准库的排序函数会接收一个比较器函数（comparator function）指针，用来判断两条数据是大于、小于还是等于。

qsort()函数看起来像这样：

```
qsort(void *array,
      size_t length,
      size_t item_size,
      int (*compar)(const void *, const void *));
```

这是一个数组指针。

这是数组长度。

这是数组中每个元素的长度。

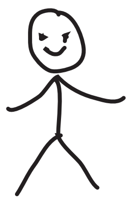
别忘了，void*指针可以指向任何数据类型。

用来比较数组中两项数据大小的函数指针。

qsort()函数会反复比较两个数据的大小，如果顺序颠倒，计算机会交换它们。

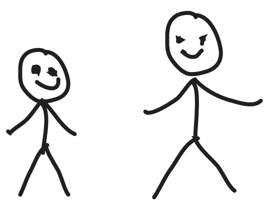
这就是为什么要使用比较器函数。它会告诉qsort()两个元素哪个排在前面，它会返回三种值：

正数



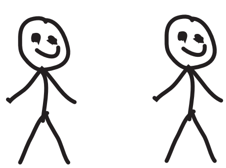
如果第一个值比第二个值大，就返回正数。

负数



如果第一个值比第二个值小，就返回负数。

0



如果两个值相等，就返回0。

下面来看一个例子，看看比较器函数在实际情况中是如何工作的。

int排序聚焦

假设有一个整型数组，你想升序排列它们，比较器函数应该长什么样子？

```
int scores[] = {543,323,32,554,11,3,112};
```

你观察qsort()接收的比较器函数的签名，会发现它接收两个**void***，也就是两个void指针。我们在使用malloc()时碰到过它，void指针可以保存**任何类型数据**的地址，但使用前必须把它转换为具体类型。

void指针（void*）可以保存任何类型的指针。

qsort()函数会两两比较数组元素，然后以正确的顺序排列它们。qsort()通过调用传给它的比较器函数来比较两个元素的大小。

```
int compare_scores(const void* score_a, const void* score_b)
{
  ...
}
```

值以指针的形式传给函数，因此要做的第一件事就是从指针中提取整型值。

```
int a = *(int*)score_a;
int b = *(int*)score_b;
```

你需要把void指针转换为整型指针。

第一个*就能得到保存在地址score_b中的整型值了。

如果a大于b，需要返回正数；如果a小于b，就返回负数；如果相等，返回0值。对整型来讲这很简单，只要将两数相减就行了：

```
return a - b;
```

如果a>b，就是正数；如果a<b，就是负数；如果a、b相等，就是0。

下面是用qsort()排序这个数组的方法：

```
qsort(scores, 7, sizeof(int), compare_scores);
```

现在轮到你了，下面描述了几种不同的排序，你能为每种排序编写比较器函数吗？为了帮你，第一个比较器函数已经写好了。

升序排列整型得分。

```
int compare_scores(const void* score_a, const void* score_b)
{
  int a = *(int*)score_a;
  int b = *(int*)score_b;
  return a - b;
}
```

降序排列整型得分。

```
int compare_scores_desc(const void* score_a, const void* score_b)
{
......................................................................
......................................................................
......................................................................
}
```

按面积从小到大排列矩形。

```
typedef struct {    <- 矩形类型。
  int width;
  int height;
} rectangle;

int compare_areas(const void* a, const void* b)
{
......................................................................
......................................................................
......................................................................
......................................................................
......................................................................
}
```

警告：这题真的很难。

按字母序排列名字，区分大小写。

```
int compare_names(const void* a, const void* b)
{
........................................................
........................................................
........................................................
}
```

友情提示：strcmp("Abc", "Def")<0

字符串是字符指针，指向字符串的指针又是什么呢？

最后，假设你已经有了compare_areas()和compare_names()，下面两个比较器函数你会怎么写？

按面积从大到小排列矩形。

```
int compare_areas_desc(const void* a, void* b)
{
........................................................
}
```

按逆字母序排列名字，区分大小写。

```
int compare_names_desc(const void* a, const void* b)
{
........................................................
}
```

练习解答

现在轮到你了，下面描述了几种不同的排序。请为每种排序编写比较器函数。

升序排列整型得分。

```
int compare_scores(const void* score_a, const void* score_b)
{
  int a = *(int*)score_a;
  int b = *(int*)score_b;
  return a - b;
}
```

这是之前已经写好的函数。

降序排列整型得分。

```
int compare_scores_desc(const void* score_a, const void* score_b)
{
  int a = *(int*)score_a;
  int b = *(int*)score_b;
  return b - a;
}
```

如果用第二个数减第一个，可以反转最终的排序。

按面积从小到大排列矩形。

```
typedef struct {
  int width;
  int height;
} rectangle;
```

矩形类型。

```
int compare_areas(const void* a, const void* b)
{
  rectangle* ra = (rectangle*)a;
  rectangle* rb = (rectangle*)b;
  int area_a = (ra->width * ra->height);
  int area_b = (rb->width * rb->height);
  return area_a - area_b;
}
```

首先，把指针转化为相应类型。

然后，计算矩形面积。

然后返回两个面积之差。

按字母序排列名字，区分大小写。

```
int compare_names(const void* a, const void* b)
{
  char** sa = (char**)a;
  char** sb = (char**)b;
  return strcmp(*sa, *sb);
}
```

字符串是字符指针，所以得到的是指针的指针。

我们要用*运算符取得字符串。

友情提示：strcmp("Abc", "Def")<0

最后，在已经有了compare_areas()和compare_names()的前提下，下面两个比较器函数你会怎么写？

按面积从大到小排列矩形。

```
int compare_areas_desc(const void* a, const void* b)
{
  return compare_areas(b, a);
}
```

或者也可以写 -compare_areas(a, b)。

按逆字母序排列名字，区分大小写。

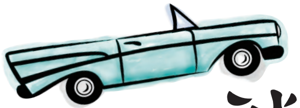
```
int compare_names_desc(const void* a, const void* b)
{
  return compare_names(b, a);
}
```

或者也可以写 -compare_names(a, b)。

轻松一刻

即使你被这题难倒了也不用担心。

本题涉及指针、函数指针，甚至还有一些数学，如果你觉得很难，那就休息一下，喝两口水，过一两小时以后再来试试。



试驾

有几个比较器函数确实比较难写，有必要运行一下。需要用以下代码调用比较器函数。

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
```

比较器函数放在这里。

```
int main()
{
  int scores[] = {543,323,32,554,11,3,112};
  int i;
  qsort(scores, 7, sizeof(int), compare_scores_desc);
  puts("These are the scores in order:");
  for (i = 0; i < 7; i++) {
    printf("Score = %i\n", scores[i]);
  }
  char *names[] = {"Karen", "Mark", "Brett", "Molly"};
  qsort(names, 4, sizeof(char*), compare_names);
  puts("These are the names in order:");
  for (i = 0; i < 4; i++) {
    printf("%s\n", names[i]);
  }
  return 0;
}
```

这行代码对得分进行排序。

qsort()改变了数组元素的顺序。

输出排序后的数组。

排序名字。

别忘了，名字数组是一个字符指针数组，因此每一项的大小是sizeof(char*)。

打印排序以后的名字。

编译并运行代码，将得到：

```
File Edit Window Help Sorted
> ./test_drive
These are the scores in order:
Score = 554
Score = 543
Score = 323
Score = 112
Score = 32
Score = 11
Score = 3
These are the names in order:
Brett
Karen
Mark
Molly
>
```

太棒了，代码工作了！

现在试着写一些你自己的代码吧。排序函数很有用，比较器函数却很难写，不过熟能生巧，多写几个就会了。

动手吧！

这里没有蠢问题

问： 用来给字符串数组排序的比较器函数使用了char**，它是什么意思？

答： 字符串数组中的每一项都是字符指针（char*），当qsort()调用比较器函数时，会发送两个指向数组元素的指针，也就是说比较器函数接收到的是指向字符指针的指针，在C语言中就是char**。

问： 当调用strcmp()时，为什么是strcmp(*a, *b)而不是strcmp(a, b)？

答： a、b的类型是char**，而strcmp()函数需要接收char*类型的值。

问： qsort()会创建新数组吗？

答： 不会，qsort()在原数组上进行改动。

问： 为什么我的头有点疼？

答： 别担心，指针很难用，如果你一点儿也不感到困扰，可能是想得还不够深。

分手信自动生成器

假设你在写一个群发邮件的程序，向不同人发送不同类型的消息，一种创建回复数据的方法是使用结构：

可能发送三类消息。

```
enum response_type {DUMP, SECOND_CHANCE, MARRIAGE};
typedef struct {
  char *name;
  enum response_type type;
} response;
```

在每条回复数据中记录回复类型。

你将发送三种类型的回复，每条回复都要保存回复类型，回复类型用枚举表示。在使用新数据类型response时需要根据回复类型分别调用以下三个函数：

```
void dump(response r)
{
  printf("Dear %s,\n", r.name);
  puts("Unfortunately your last date contacted us to");
  puts("say that they will not be seeing you again");
}

void second_chance(response r)
{
  printf("Dear %s,\n", r.name);
  puts("Good news: your last date has asked us to");
  puts("arrange another meeting. Please call ASAP.");
}

void marriage(response r)
{
  printf("Dear %s,\n", r.name);
  puts("Congratulations! Your last date has contacted");
  puts("us with a proposal of marriage.");
}
```

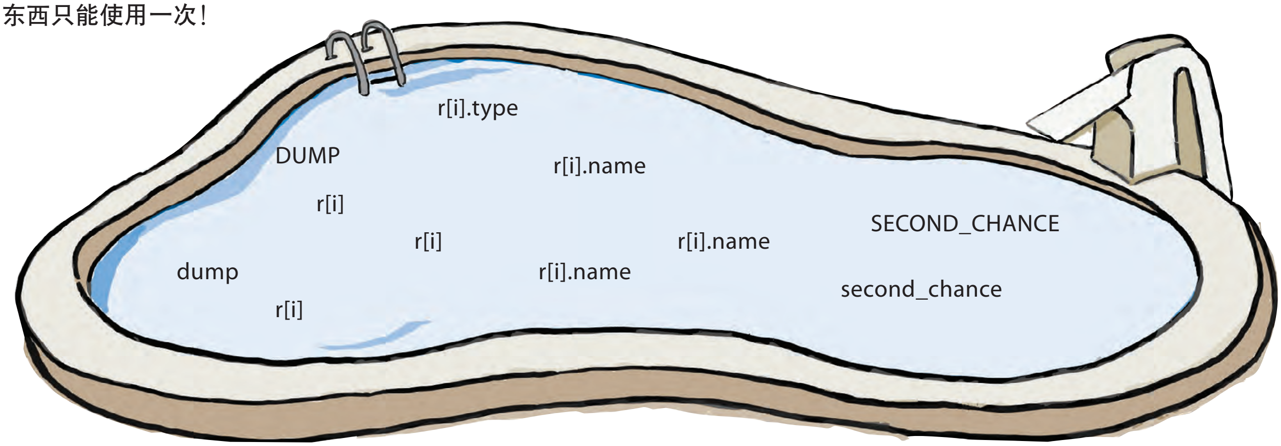
你已经有了数据结构，生成回复的函数也有了，下面就来看看如何根据response数组批量生成回复。

游泳池拼图

从游泳池中取出代码片段，放到下面的空白横线处。你的目标是拼凑出main()函数，为response数组批量生成邮件。每个片段最多只能使用一次。

```
int main()
{
  response r[] = {
    {"Mike", DUMP}, {"Luis", SECOND_CHANCE},
    {"Matt", SECOND_CHANCE}, {"William", MARRIAGE}
  };
  int i;
  for (i = 0; i < 4; i++) {
    switch(........................) {
    case ........................:
      dump(........................);
      break;
    case ........................:
      second_chance(........................);
      break;
    default:
      marriage(........................);
    }
  }
  return 0;
}
```

注意：游泳池中的每样东西只能使用一次！



泳池拼图解答

从游泳池中取出代码片段，放到下面的空白横线处。你的目标是拼凑出main()函数，为response数组批量生成邮件。每个片段最多只能使用一次。

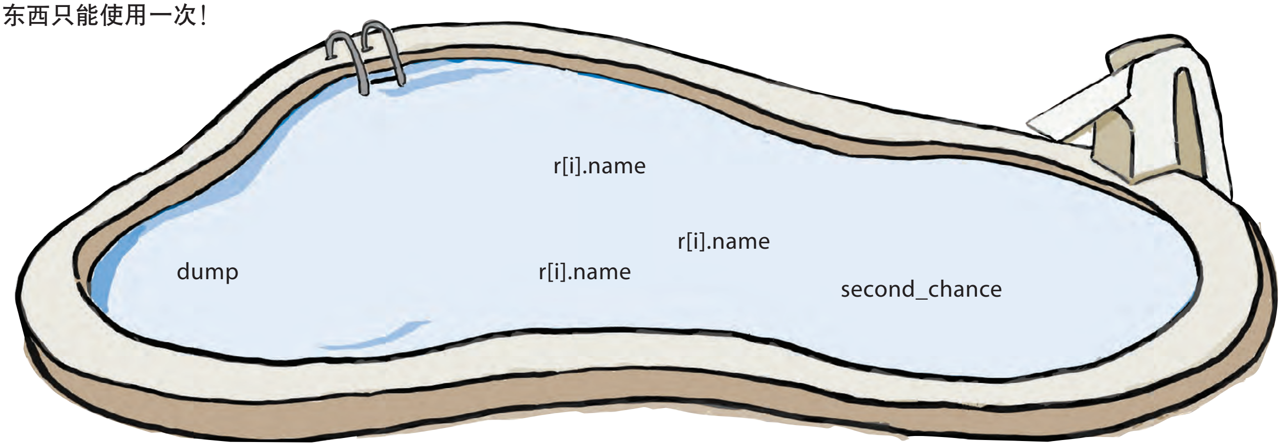
```
int main()
{
  response r[] = {
    {"Mike", DUMP}, {"Luis", SECOND_CHANCE},
    {"Matt", SECOND_CHANCE}, {"William", MARRIAGE}
  };
  int i;
  for (i = 0; i < 4; i++) {
    switch(r[i].type) {
    case DUMP:
      dump(r[i]);
      break;
    case SECOND_CHANCE:
      second_chance(r[i]);
      break;
    default:
      marriage(r[i]);
    }
  }
  return 0;
}
```

循环遍历数组。

每次都要检查type字段。

根据类型调用相应方法。

注意：游泳池中的每样东西只能使用一次！



试驾

当运行程序时，程序果然为每个人都生成了相应的回复：

```
File Edit Window Help DontForgetToBreak
./send_dear_johns
Dear Mike,
Unfortunately your last date contacted us to
say that they will not be seeing you again
Dear Luis,
Good news: your last date has asked us to
arrange another meeting. Please call ASAP.
Dear Matt,
Good news: your last date has asked us to
arrange another meeting. Please call ASAP.
Dear William,
Congratulations! Your last date has contacted
us with a proposal of marriage.
>
```

程序正确运行了，但代码中充斥着大量函数调用，每次都需要根据回复类型来调用函数，看起来像这样：

```
switch(r.type) {
case DUMP:
  dump(r);
  break;
case SECOND_CHANCE:
  second_chance(r);
  break;
default:
  marriage(r);
}
```

如果增加第四种回复类型，你就不得不修改程序中每一个像这样的地方。很快，就有一大堆代码需要维护，而且这样很容易出错。

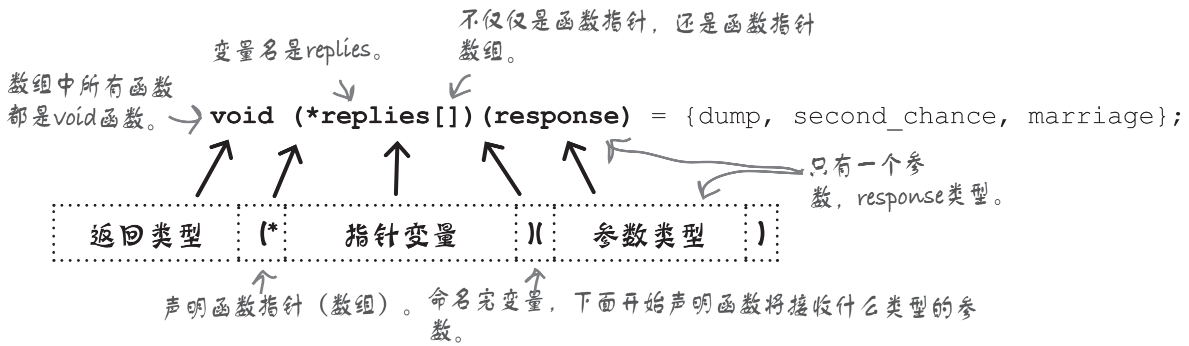
好在可以使用一个C语言的技巧，这个技巧涉及数组……

创建函数指针数组

这个技巧就是创建一个与回复类型一一对应的函数指针数组。在此之前，我们先看看怎么创建函数指针数组。如果想在数组中保存一组函数名，可以这样写：

```
replies[] = {dump, second_chance, marriage};
```

但这样的语法在C语言中行不通，如果想在数组中保存函数，就必须告诉编译器函数的具体特征：函数返回什么类型以及接收什么参数。也就是说必须使用下面这种**复杂得多**的语法：



如何用数组解决刚才的问题？

观察数组，函数名的顺序与枚举类型的顺序完全相同：

```
enum response_type {DUMP, SECOND_CHANCE, MARRIAGE};
```

这点很重要，因为当C语言在创建枚举时会给每个符号分配一个从0开始的数字，所以`DUMP == 0`，`SECOND_CHANCE == 1`，而`MARRIAGE == 2`，也就是说可以通过**response_type**获取数组中的函数指针。

这是replies函数数组。

```
replies[SECOND_CHANCE] == second_chance
```

等价于second_chance的函数名。

SECOND_CHANCE的值是1。

能否用函数数组来修改之前的main()函数呢？

磨笔上阵

虽然这道题很难，但只要多花点时间应该没什么问题。补全这段代码所需的知识你都已经掌握了。在新版main()函数中，switch/case语句已移除，你需要用一行代码来替代它，这行代码将从replies数组中找到对应的函数名，然后用它来调用函数。

```
void (*replies[])(response) = {dump, second_chance, marriage};

int main()
{
  response r[] = {
    {"Mike", DUMP}, {"Luis", SECOND_CHANCE},
    {"Matt", SECOND_CHANCE}, {"William", MARRIAGE}
  };
  int i;
  for (i = 0; i < 4; i++) {
    ..................................................................
  }
  return 0;
}
```

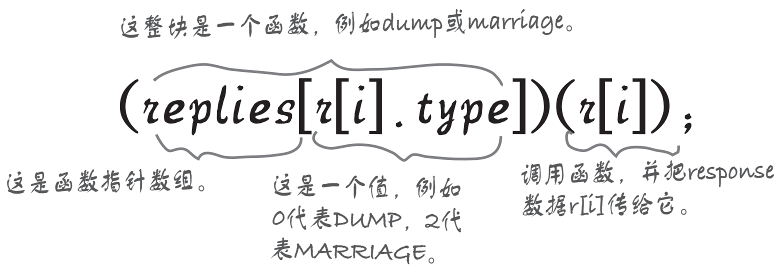
磨笔上阵 解答

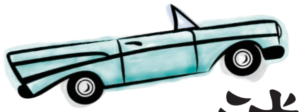
这道题很难，新版main()函数中，switch/case语句已移除，你需要用一行代码来替代它，这行代码将从replies数组中找到相应函数名，然后用它来调用函数。

```
void (*replies[])(response) = {dump, second_chance, marriage};

int main()
{
  response r[] = {
    {"Mike", DUMP}, {"Luis", SECOND_CHANCE},
    {"Matt", SECOND_CHANCE}, {"William", MARRIAGE}
  };
  int i;
  for (i = 0; i < 4; i++) {
    (replies[r[i].type])(r[i]);
  }
  return 0;
}
```

你也可以在第一个开括号后面加上*，效果相同。

我们来分解这个表达式。





试驾

当运行新版程序时，得到了和刚才一样的输出：

```
File Edit Window Help WhoIsJohn
> ./dear_johns
Dear Mike,
Unfortunately your last date contacted us to
say that they will not be seeing you again
Dear Luis,
Good news: your last date has asked us to
arrange another meeting. Please call ASAP.
Dear Matt,
Good news: your last date has asked us to
arrange another meeting. Please call ASAP.
Dear William,
Congratulations! Your last date has contacted
us with a proposal of marriage.
>
```

区别呢？现在你用下面这行代码代替了整个switch语句：

```
(replies[r[i].type])(r[i]);
```

如果需要在程序中多次调用回复函数，你不必复制很多代码，而当决定添加新的回复类型和函数时，只需要把它加到数组中即可：

```
enum response_type {DUMP, SECOND_CHANCE, MARRIAGE, LAW_SUIT};
void (*replies[])(response) = {dump, second_chance, marriage, law_suit};
```

可以像这样添加新的回复类型和函数。

函数指针数组让代码易于管理，它们让代码变得更短、更易于扩展，从而可以伸缩。一开始理解起来有些费劲，但函数指针数组的确可以提高C编程技巧。

要点

- 函数指针中保存了函数的地址。
- 函数名其实是函数指针。②
- 如果你有函数shoot()，那么shoot和&shoot都指向了shoot()函数。
- 可以用“返回类型(*变量名)(参数类型)”来声明新的函数指针。
- 如果fp是函数指针，那么可以用fp(参数，……)调用函数。
- 也可以用(*fp)(参数，……)，两种情况都能工作。
- C标准库中有一个叫qsort()的排序函数。
- qsort()接收指向比较器函数的指针，比较器函数可以比较两个值的大小。
- 比较器函数接收两个指针，分别指向待排序数组中的两项。
- 如果把数据保存在数组中，就可以用函数指针数组将函数与数据项关联起来。

这里没有蠢问题

问： 为什么函数指针的语法这么复杂？

答： 因为当声明函数指针时，需要说明返回类型和参数类型，这就解释了为什么有那么多的括号。

问： 刚才那段代码看起来有点像其他语言中面向对象的代码，是吗？

答： 的确很像，面向对象语言将一组函数（称为方法）与数据关联在一起。同样你也可以用函数指针将函数与数据关联在一起。

问： 也就是说C语言也是面向对象的？太好了。

答： C语言不是面向对象语言，不过一些以C语言为基础的语言，例如Objective-C和C++，在底层使用函数指针时创建了很多面向对象的特性。

② 并不完全等于，函数名是L-value，在存储器中不分配变量。——译者注

让函数能伸能缩

你既需要功能强大如find()那样可以用函数指针进行搜索的函数，也需要易于使用的函数。printf()函数有一个很酷的功能：接收参数的数量可变。

```
printf("%i bottles of beer on the wall, %i bottles of beer\n", 99, 99);
printf("Take one down and pass it around, ");
printf("%i bottles of beer on the wall\n", 98);
```

想打印几个参数，就可以传给printf()几个。

你的函数如何做到这点？

这里刚好有个问题需要用到可变数量参数。Head First酒吧里的人正在为计算账单而烦恼，一名员工为了提高工作效率，根据现存鸡尾酒清单创建了一个枚举类型和一个返回每种酒价格的函数：

```
enum drink {
  MUDSLIDE, FUZZY_NAVEL, MONKEY_GLAND, ZOMBIE
};

double price(enum drink d)
{
  switch(d) {
  case MUDSLIDE:
    return 6.79;
  case FUZZY_NAVEL:
    return 5.31;
  case MONKEY_GLAND:
    return 4.82;
  case ZOMBIE:
    return 5.89;
  }
  return 0;
}
```

很酷，如果Head First酒吧的员工想知道某种酒的单价，只要调用这个函数就行了。但如果他们想要计算一单酒的总价：

简单 → `price(ZOMBIE)`

`total(3, ZOMBIE, MONKEY_GLAND, FUZZY_NAVEL)` ← 没那么简单

酒的杯数

酒单列表

他们希望有一个叫total()的函数，它接收酒的杯数和这些酒的名字。

可变参数函数

参数数量可变的函数被称为**可变参数函数**（variadic function）。C标准库中有一组**宏**（macro）可以帮助你建立自己的可变参数函数。为了弄清它是如何工作的，你将创建一个函数打印一连串int的函数：

可以把宏想象成一种特殊类型的函数，它可以修改源代码。

```
print_ints(3, 79, 101, 32);
```

要打印几个int。

要打印的int。

下面是代码：

```
#include <stdarg.h>

void print_ints(int args, ...)
{
  va_list ap;
  va_start(ap, args);
  int i;
  for (i = 0; i < args; i++) {
    printf("argument: %i\n", va_arg(ap, int));
  }
  va_end(ap);
}
```

可变参数跟在后面。

从args参数开始后面都是可变参数。

这是普通参数。

va_start表示可变参数从哪里开始。

循环遍历所有其他参数。

args中保存了变量的数目。

我们逐行分析。

1 包含stdarg.h头文件。
所有处理可变参数函数的代码都在*stdarg.h*中，请务必包含这个头文件。

2 告诉函数还有更多参数……
有没有看过这样的书？女英雄把男子拖进卧室，然后这一章就以“……”结束了。“……”叫省略号，它告诉你后面还有其他东西。在C语言中，函数参数后的省略号“…”表示还有更多参数。

我们也没看过。

3 创建va_list。
`va_list`用来保存传给函数的其他参数。

4 说明可变参数从哪里开始。
需要把最后一个普通参数的名字告诉C，在这个例子中就是`args`变量。

5 然后逐一读取可变参数。
参数现在全保存在`va_list`中，可以用`va_arg`读取它们。`va_arg`接收两个值：`va_list`和要读取参数的类型。本例中所有参数都是`int`。

6 最后……销毁va_list。
当读完了所有参数，要用`va_end`宏告诉C你做完了。

7 现在可以调用函数了。
一旦完成了函数，就可以调用它：

```
print_ints(3, 79, 101, 32);
```

将打印79，101和32。

百宝箱

函数与宏

宏用来在编译前重写代码，这里的几个宏va_start、va_arg和va_end看起来很像函数，但实际上隐藏在它们背后的是一些神秘的指令。在编译前，预处理器会根据这些指令在程序中插入巧妙的代码。

这里没有蠢问题

问： 等等，为什么va_end和va_start叫宏？它们不就是一般的函数吗？

答： 不是，它们只是设计成了普通函数的样子，预处理会把它们替换成其他代码。

问： 什么是预处理器？

答： 预处理器在编译阶段之前运行，它会做很多事情，包括把头文件包含进代码。

问： 可以只使用可变参数，而不用普通参数吗？

答： 不行，至少需要一个普通参数，只有这样才能把它的名字传给va_start。

问： 如果我从va_arg中读取比传给函数更多的参数会怎样？

答： 会发生不确定的错误。

问： 听起来真糟糕。

答： 是的，非常糟糕。

问： 如果我以double或其他类型读取int参数呢？

答： 也会发生不确定的错误。

下面轮到你上场了，Head First酒吧的人想要创建一个函数，能够返回一巡酒的总价，函数如下：

```
printf("Price is %.2f\n", total(3, MONKEY_GLAND, MUDSLIDE, FUZZY_NAVEL));
```

将打印"Price is 16.92"。

使用前几页上的`price()`函数完成`total()`函数的代码：

```
double total(int args, ...)
{
  double total = 0;
  ......................................................
  ......................................................
  ......................................................
  ......................................................
  ......................................................
  ......................................................
  ......................................................
  ......................................................
  return total;
}
```

下面轮到你上场了，Head First酒吧的人想要创建一个函数，能够返回一巡酒的总价，函数如下：

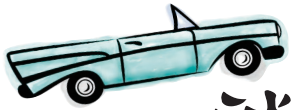
```
printf("Price is %.2f\n", total(3, MONKEY_GLAND, MUDSLIDE, FUZZY_NAVEL));
```

将打印“Price is 16.92”。

使用前几页上的price()函数完成total()函数的代码：

即使你的代码和这里的不完全一样也没关系，有好几种写法。

```
double total(int args, ...)
{
  double total = 0;
  va_list ap;
  va_start(ap, args);
  int i;
  for(i = 0; i < args; i++) {
    enum drink d = va_arg(ap, enum drink);
    total = total + price(d);
  }
  va_end(ap);
  return total;
}
```



试驾

你创建了一些调用函数的测试代码，编译代码并查看结果：

这是测试代码。

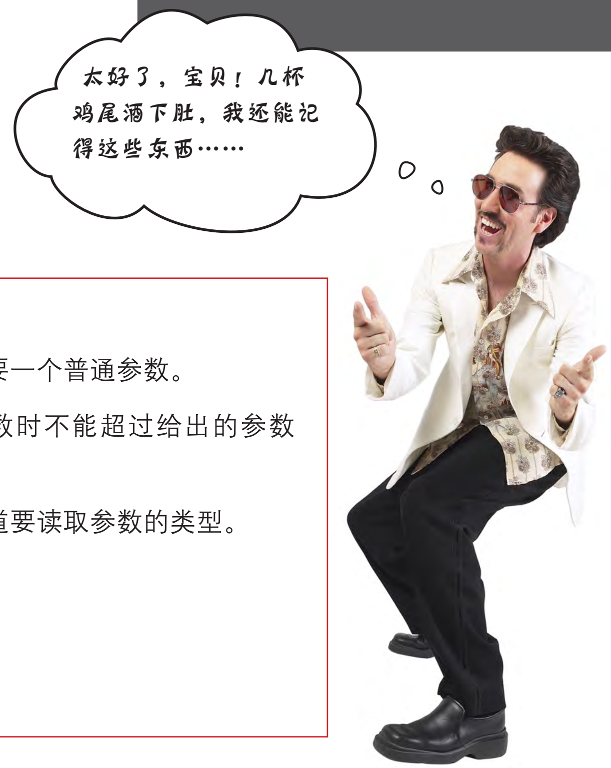
```
main(){
  printf("Price is %.2f\n", total(2, MONKEY_GLAND, MUDSLIDE));
  printf("Price is %.2f\n", total(3, MONKEY_GLAND, MUDSLIDE, FUZZY_NAVEL));
  printf("Price is %.2f\n", total(1, ZOMBIE));
  return 0;
}
```

```
File Edit Window Help Cheers
> ./price_drinks
Price is 11.61
Price is 16.92
Price is 5.89
>
```

这是输出。

代码正确运行了！

现在你学会了使用可变参数，代码用起来更简单，也更直观了。



要点

- 接收数量可变参数的函数叫可变参数函数。
- 为了创建可变参数函数，需要包含*stdarg.h*头文件。
- 可变参数将保存在va_list中。
- 可以用va_start()、va_arg()和va_end()控制va_list。
- 至少需要一个普通参数。
- 读取参数时不能超过给出的参数个数。
- 需要知道要读取参数的类型。

C语言工具箱

你已经学完了第7章，现在你的工具箱又加入了高级函数。关于本书的提示工具条的完整列表，请见附录ii。

函数指针是唯一不需要加*和&运算符的指针……

……只要你想，也可以加上它们。

有了函数指针，就可以把函数当数据传递。

每个函数的名字都是一个指向函数的指针。

qsort()会排序数组。

排序函数接收比较器函数指针。

参数数量可变的函数叫“可变参数函数”。

包含stdarg.h，就可以创建可变参数函数。

比较器函数决定如何排序两条数据。

有了函数指针数组，就可以根据不同类型的数据运行不同的函数。

8 静态库与动态库

热插拔代码

你已经见识过标准库的威力。

是时候在代码中发挥这种威力了。在本章中，你将学会创建自己的库，以及在多个程序中复用相同代码；还将掌握编程大师的秘诀——通过动态库在运行时共享代码；最后你将写出易于扩展并可以有效管理的代码。

值得信赖的代码

还记得encrypt()函数吗？你用它加密过字符串。encrypt()放在一个单独的源文件中，这样就能在多个程序中使用它：

```
#include "encrypt.h"

void encrypt(char *message)
{
  while (*message) {
    *message = *message ^ 31;
    message++;
  }
}
```

encrypt.c

```
void encrypt(char *message);
```

encrypt.h

某人写了一个叫checksum()的函数，它可以用来校验字符串是否被篡改。数据的加密与防篡改是安全领域两个很重要的问题。既然这两个函数都那么有用，我们就用它们来构建一个安全库。

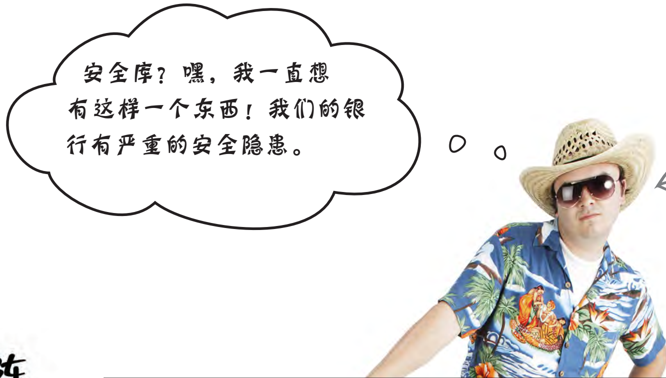
```
#include "checksum.h"

int checksum(char *message)
{
  int c = 0;
  while (*message) {
    c += c ^ (int)(*message);
    message++;
  }
  return c;
}
```

函数根据字符串的内容，返回一个数字。

checksum.c

```
int checksum(char *message);
```

checksum.h



磨笔上阵

为了检查函数能否工作，银行的工作人员写了一个测试程序。他把所有源文件放在同一个目录下，然后编译程序。

他先把两个安全文件编译为目标文件，然后写了一个测试程序：

```
#include <stdio.h>
#include <encrypt.h>
#include <checksum.h>

int main()
{
  char s[] = "Speak friend and enter";
  encrypt(s);
  printf("Encrypted to '%s'\n", s);
  printf("Checksum is %i\n", checksum(s));
  encrypt(s);
  printf("Decrypted back to '%s'\n", s);
  printf("Checksum is %i\n", checksum(s));
  return 0;
}
```

```
File Edit Window Help
> gcc -c encrypt.c -o encrypt.o
> gcc -c checksum.c -o checksum.o
>
```

encrypt()会加密你的数据，再调用一次就会解密数据。

问题发生了，编译程序时出了错……

```
File Edit Window Help
> gcc test_code.c encrypt.o checksum.o -o test_code
test_code.c:2:21: error: encrypt.h: No such file or directory
test_code.c:3:22: error: checksum.h: No such file or directory
>
```

请用铅笔画出导致编译错误的代码或命令。

磨笔上阵 解答

问题出在测试程序中，所有源文件都保存在同一个目录下，测试程序却用尖括号（<>）包含了头文件*encrypt.h*和*checksum.h*。

```
#include <stdio.h>
#include <encrypt.h>
#include <checksum.h>

int main()
{
  char s[] = "Speak friend and enter";
  encrypt(s);
  printf("Encrypted to '%s'\n", s);
  printf("Checksum is %i\n", checksum(s));
  encrypt(s);
  printf("Decrypted back to '%s'\n", s);
  printf("Checksum is %i\n", checksum(s));
  return 0;
}
```

尖括号代表标准头文件

如果在#include语句中使用尖括号，编译器就会在**标准**头文件目录中查找头文件，而不是当前目录。

为了用**本地**头文件编译程序，需要把尖括号换成双引号（""）：

stdio.h保存在某个标准头文件目录下。

```
#include <stdio.h>
#include "encrypt.h"
#include "checksum.h"
```

encrypt.h、checksum.h和程序在同一个目录中。

现在代码正确编译，它把测试字符串加密为了一段乱码。

```
File Edit Window Help <>
> gcc test_code.c encrypt.o checksum.o -o test_code
> ./test_code
Encrypted to 'Loz~t?ymvzq{?~q{?zqkzm'
Checksum is 89561741
Decrypted back to 'Speak friend and enter'
Checksum is 89548156
>
```

checksum会根据不同字符串返回不同值。

再调用一次encrypt()函数，就会返回原字符串。

标准头文件目录在哪里？

如果用尖括号包含了头文件，编译器会去哪里找头文件？为了找到这个问题的答案，需要查看编译器自带的文档。通常类UNIX操作系统（如Mac或Linux）中，编译器会在以下目录查找头文件：

```
/usr/local/include
/usr/include
```

编译器会先检查/usr/local/include。

/usr/local/include通常用来放第三方库的头文件。

/usr/include一般用来放操作系统的头文件。

如果你用的是MinGW版的**gcc**，编译器会在下面这个目录中查找：

```
C:\MinGW\include
```

如何共享代码？

有时你想在多个程序中使用相同代码，但这些程序四散在计算机中各个角落，在不同的文件夹中，这时该怎么办？

你希望在程序之间共享两类代码：*.h*头文件和*.o*目标文件，下面就来看看如何共享它们。

共享.h头文件

在多个C项目中共享头文件的方法很多：

1 把头文件保存在标准目录中。

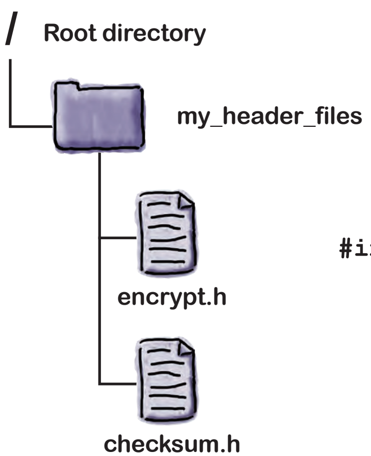
只要把头文件复制到*/usr/local/include*这样的标准目录中，就可以在源代码中用尖括号包含它们。

```
#include <encrypt.h>
```

只要头文件在标准目录中，就可以用尖括号包含它们。

2 在include语句中使用完整路径名。

如果你想把头文件放在其他地方，如*/my_header_files*，可以把目录名加到`include`语句中：




```
#include "/my_header_files/encrypt.h"
```

3 你可以告诉编译器去哪里找头文件。

最后一种方法是告诉编译器去哪里找头文件，可以使用`gcc`的`-I`选项：

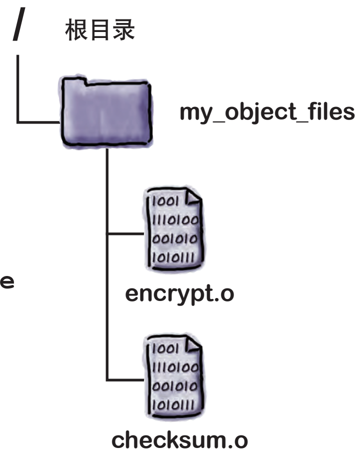
```
gcc -I/my_header_files test_code.c ... -o test_code
```

让编译器同时在/my_header_files和标准目录中进行查找。

`-I`选项告诉`gcc`编译器还可以去哪里找头文件。编译器会先检查`-I`选项中的目录，然后像往常一样检查所有标准目录。

用完整路径名共享.o目标文件

可以把.*o*目标文件放在一个类似共享目录的地方。当编译程序时，只要在目标文件前加上完整路径就行了：

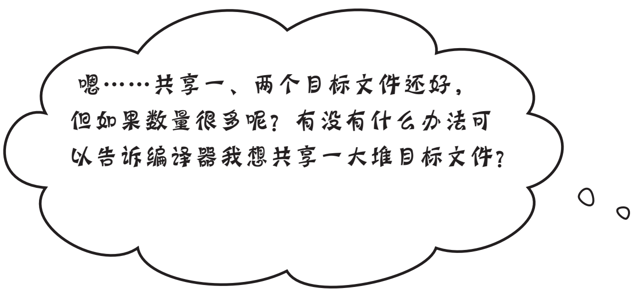



```
gcc -I/my_header_files test_code.c
      /my_object_files/encrypt.o
      /my_object_files/checksum.o -o test_code
```

使用目标文件的完整路径名，你就能在多个C项目中共享它们。

/my_object_files就好比一个中央仓库，专门用来保存目标文件。

只要在编译代码时使用目标文件的完整路径名，所有C程序都能共享*encrypt.o*和*checksum.o*文件。



只要创建目标文件存档，就可以一次告诉编译器一批目标文件。

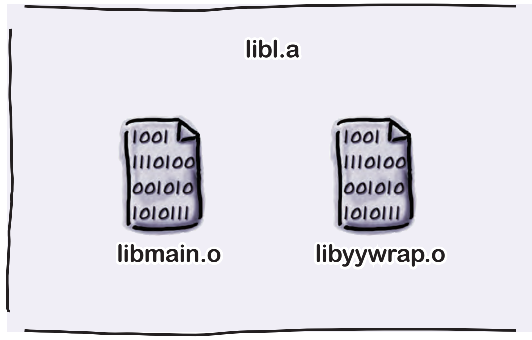
把一批目标文件打包在一起就成了存档文件。创建安全代码的存档文件，就可以很方便地在多个项目之间共享代码。

我们来看看怎么做……

存档中包含多个.o文件

如果你用过*.zip*或*.tar*文件，就知道创建一个包含其他文件的文件是一件多么容易的事。

打开终端或命令提示符，进入某个库目录，比如*/usr/lib*或*C:\MinGW\lib*，库代码就放在这些目录下。你可以在库目录中看到一大批*.a*存档，你可以用nm命令查看存档中的内容：



你的计算机上可能没有libl.a，但可以用nm命令查看其他.a文件的内容。

这个存档叫libl.a。

libmain.o

libyywrap.o

"T_main"表明libmain.o包含main()函数。

```
File Edit Window Help SilenceInTheLibrary
> nm libl.a

libl.a(libmain.o):
00000000000003a8 s EH_frame0
                 U _exit
0000000000000000 T _main
00000000000003c0 S _main.eh
                 U _yylex

libl.a(libyywrap.o):
0000000000000350 s EH_frame0
0000000000000000 T _yywrap
0000000000000368 S _yywrap.eh
>
```

nm命令列出了存档中保存文件的名字。*libl.a*有两个目标文件：*libmain.o*和*libyywrap.o*。别管它们是做什么的，这个例子只是为了说明可以把一批目标文件转化为存档，然后在gcc中使用。

在学习怎样用*.a*文件编译程序之前，先看看如何在存档中保存*encrypt.o*和*checksum.o*文件。

用ar命令创建存档

存档命令（**ar**）会在存档文件中保存一批目标文件：

r表示如果.a文件存在就更新它。

s告诉ar要在.a文件开头建立索引。

这些是将保存在存档中的文件。

```
ar -rcs libhfsecurity.a encrypt.o checksum.o
```

c表示创建存档时不显示反馈信息。

这是要创建的.a文件名。

注意到没有？所有*.a*文件名都是***libXXX.a***的形式。这是命名存档的标准方式，存档是**静态库**（static library），所以要以*lib*开头，稍后你会看到什么是**静态库**。

务必把存档命名为libXXX.a。

否则编译器找不到它们。

……在库目录下保存.a文件

你可以把存档保存在库目录中，用哪个库目录由你做主，有以下两种选择：

- **把.a文件保存在标准目录中，如/usr/local/lib。**
  有的程序员在确保他们的代码能正确运行以后就会把存档安装在标准目录中。在Linux、Mac与Cygwin中，可以把存档保存在*/usr/local/lib*中，这个目录专门用来放本地自定义库。

- **把.a文件放在其他目录中。**
  如果你还处于开发阶段，或者在系统目录中安装代码让你觉得很不爽，也可以创建自己的库目录，例如：*/my_lib*。

在很多机器上，只有系统管理员才能把文件放到/usr/local/lib中。

最后编译其他程序

创建库存档是为了能在其他程序中使用它，当你把存档安装到标准目录后，就可以用-l开关编译代码：

别忘了，在用-l选项包含库之前列出源文件。

hfsecurity叫编译器去找一个叫libhfsecurity.a的存档。

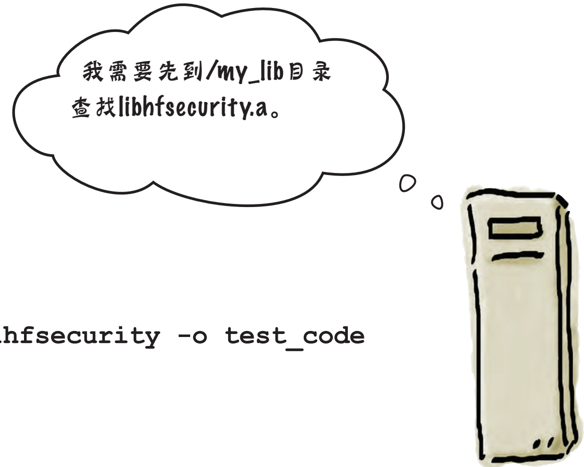
```
gcc test_code.c -lhfsecurity -o test_code
```

是否需要使用-I选项取决于把头文件放在了哪里。

如果要使用多个存档，可以设置多个-l选项。

现在知道为什么要把存档命名为*libXXX.a*了吧。-l选项后的名字必须与存档名的一部分匹配。如果你的存档叫*libawesome.a*，可以用-lawesome开关编译程序。

如果想把存档放在其他地方呢？比如*/my_lib*。你可以用-L选项告诉编译器去哪个目录查找存档：




```
gcc test_code.c -L/my_lib -lhfsecurity -o test_code
```

百宝箱

为什么不同机器库目录的内容相差这么多？因为不同操作系统提供了不同的服务。每个*.a*文件都是一个独立的库，有的库用来连接网络，有的用来创建GUI程序。

我们找几个*.a*文件来试用一下nm命令。每个模块都列出了很多名字，它们是一些已经编译好了的函数，你可以在程序中使用它们：

T代表“文本（Text）”，说明它是一个函数。

```
0000000000000000 T _yywrap
```

函数名是yywrap()。

nm命令会告诉你每个*.o*目标文件的名字，然后列出所有目标文件中的名字，如果某个名字前出现了**T**，就说明它是目标文件中某个函数的名字。

Make冰箱贴

安保人员在使用新版安全库编译银行程序时遇到了问题。他把自己的源代码和encrypt、checksum的源代码放在了同一个目录下，他想在这个目录中创建*libhfsecurity.a*存档，然后用它来编译程序，你能帮他补全makefile吗？

注意：bank_vault程序用了下面的#include语句：

```
#include <encrypt.h>
#include <checksum.h>
```

这是makefile:

```
encrypt.o: encrypt.c
        gcc ............................. encrypt.c -o encrypt.o

checksum.o: checksum.c
        gcc ............................. checksum.c -o checksum.o

libhfsecurity.a: encrypt.o .............................
        ar -rcs ............................. encrypt.o .............................

bank_vault: bank_vault.c .............................
        gcc ............................. -I ............. -L ............. ............................. -o bank_vault
```

-c

-c

checksum.o

libhfsecurity.a

bank_vault.c

checksum.o

-lhfsecurity

.

libhfsecurity.a

/usr/lib

/usr/local/include

.

-rcs

/usr/local/lib

-rcs

Make冰箱贴解答

安保人员在使用新版安全库编译银行程序时遇到了问题。他把自己的源代码和encrypt、checksum的源代码放在了同一个目录下，他想在这个目录中创建*libhfsecurity.a*存档，然后用它来编译程序，你将帮助他补全makefile。

注意：bank_vault程序用了下面的#include语句：

```
#include <encrypt.h>
#include <checksum.h>
```

#include语句使用了尖括号。要用-I语句告诉编译器头文件在哪里。

这是makefile：

```
encrypt.o: encrypt.c
      gcc -c encrypt.c -o encrypt.o

checksum.o: checksum.c
      gcc -c checksum.c -o checksum.o

libhfsecurity.a: encrypt.o checksum.o
      ar -rcs libhfsecurity.a encrypt.o checksum.o

bank_vault: bank_vault.c libhfsecurity.a
      gcc bank_vault.c -I . -L . -lhfsecurity -o bank_vault
```

用encrypt.c源文件创建目标文件。

用checksum.c源文件创建目标文件。

只有先创建encrypt.o和checksum.o，才能建立libhfsecurity.a存档。

创建libhfsecurity.a存档。

需要加上-lhfsecurity，因为存档叫libhfsecurity.a。

需要在库代码前列出程序源代码。

需要-I.，因为头文件在“.”（当前）目录下。

要用-L.，因为存档在当前目录下。

/usr/lib　　/usr/local/include　　-rcs　　/usr/local/lib　　-rcs

要点

- 使用尖括号（<>），编译器就会从标准目录中读取头文件。
- 常见的标准头文件目录有*/usr/include*和*C:\MinGW\include*。
- 一个库存档中有多个目标文件。
- 可以用`ar -rcs libarchive.a file0.o file1.o...`创建存档。
- 库存档名应以*lib*开头，以*.a*结尾。
- 如果想链接一个叫*libfred.a*的存档，就使用`-lfred`选项。
- 在`gcc`命令中，`-l`标志应该在源代码文件后出现。

这里没有蠢问题

问：我怎样才能知道计算机上哪些目录是标准库目录？

答：你需要查看编译器文档。在大多数类Unix操作系统中，标准库目录有*/usr/lib*和*/usr/local/lib*。

问：我想把库存档放到*/usr/lib*目录下，但计算机不许我那么做，为什么？

答：出于安全考虑，操作系统为了防止你一不小心破坏某个库，会禁止你往标准目录中写文件。

问：`ar`命令的存档格式在所有系统中都是一样的吗？

答：不是，虽然不同平台之间存档格式区别不大，但存档中目标代码的格式在不同操作系统中可谓天差地别。

问：创建库存档以后能不能查看里面的内容？

答：可以，`ar -t`<文件名>会列出存档中的目标文件。

问：存档会像可执行文件那样把目标文件链接在一起吗？

答：不会，目标文件以独立文件的形式保存在存档中。

问：我可以把任何类型的文件放在存档中吗？

答：不可以，`ar`命令会先检查文件类型。

问：我能从存档中提取某个目标文件吗？

答：可以的，你可以使用`ar -x libhfsecurity.a encrypt.o`命令把*encrypt.o*文件从*libhfsecurity.a*中提取出来。

问：为什么要叫"静态"链接？

答：因为一旦链接以后就不能修改。静态链接就好比在咖啡中加入牛奶，混在一起就不能再分开。

问：我能用Head First安全库保护银行数据的安全吗？

答：最好不要这么做。

链接器有约

本周访谈：
你到底是做什么的？

Head First：链接器，非常感谢你能抽出时间来参加我们的节目。

链接器：我很高兴来到这里。

Head First：你有没有觉得开发人员忽视了你？他们压根不知道你是干嘛的。

链接器：我不太善于交际，很多人不会通过`ld`命令直接与我对话。

Head First：`ld`？

链接器：`ld`正是鄙人。

Head First：屏幕上显示了很多选项。

链接器：确实如此。我有很多选项，它们代表链接程序的不同方法，这就是为什么一些人只用`gcc`命令。

Head First：编译器也能链接文件吗？

链接器：编译器会制定链接方案，然后调用我。我会默默地把它们链接起来，你完全不知道我的存在。

Head First：我还有一个问题……

链接器：什么？

Head First：我知道这个问题很愚蠢，但你到底是做什么的？

链接器：我把编译后的代码缝合起来，有点儿像电话接线员做的工作。

Head First：不明白。

链接器：老式电话接线员会把两个地方的电话线路连接起来，这样两边的人才能通话，目标文件也是如此。

Head First：怎么说？

链接器：一个目标文件可能需要调用另一个目标文件中的函数，我会把这个文件中的函数调用与那个文件中的函数链接在一起。

Head First：那你一定很有耐心。

链接器：我喜欢做这种事，没事的时候我会绣十字绣。

Head First：真的啊？

链接器：开玩笑的。

Head First：谢谢你，链接器。

Head First健身房全球化战略

Head First健身房打算把业务扩展到全球范围内，他们在四大洲都开设了分店，每个分店都使用自家的“流汗流血不流泪”牌健身器材。健身房的技术人员正在为椭圆机、跑步机和健身车编写软件。软件将从设备传感器中读取数据，然后在小型LCD屏幕上显示信息，告诉用户他们跑了多少距离，消耗了多少卡路里。

计划就是这样，他们需要一些帮助，下面来看看代码的细节。

计算卡路里

技术小组还在赶代码，但他们已经有了一个核心模块。*hfcal*库负责生成LCD所需数据。只要把用户的体重、跑动距离和一个特殊系数传给代码，它就会在标准输出打印LCD信息：

```
#include <stdio.h>
#include <hfcal.h>

void display_calories(float weight, float distance, float coeff)
{
  printf("Weight: %3.2f lbs\n", weight);
  printf("Distance: %3.2f miles\n", distance);
  printf("Calories burned: %4.2f cal\n", coeff * weight * distance);
}
```

hfcal.h头文件中只有display_calories()函数的声明。

体重的单位是磅。

距离的单位是英里。

这段代码保存在一个叫hfcal.c的文件中。

hfcal.c

技术小组还没有来得及为每类器材编写代码。当他们写完以后，椭圆机、跑步机和健身车将分别会有一个程序。在此之前，他们先创建了一个测试程序，用一些测试数据来调用*hfcal.c*中的函数：

```
#include <stdio.h>
#include <hfcal.h>

int main()
{
  display_calories(115.2, 11.3, 0.79);
  return 0;
}
```

用户的体重为115.2磅，在椭圆机上跑了11.3英里。

对这台机器来说，系数是0.79。

这是测试代码。

elliptical.c

LCD显示器会捕捉标准输出中的数据。

测试程序会显示这些信息。

磨笔上阵

你已经看到了测试程序和*hfcal*库的源代码，下面就来构建代码。看你还记不记得这些命令。

1. 首先创建一个叫*hfcal.o*的目标文件，*hfcal.h*头文件将保存在*./includes*中：

2. 接着你需要用测试代码*elliptical.c*创建一个叫*elliptical.o*的目标文件：

3. 现在你需要用*hfcal.o*创建存档库，并把它保存到*./libs*：

4. 最后，用*elliptical.o*和*hfcal*存档创建`elliptical`可执行文件：

磨笔上阵 解答

你已经看到了测试程序和*hfcal*库的源代码，下面就来构建代码。

看你还记不记得这些命令。

1. 首先创建一个叫*hfcal.o*的目标文件，*hfcal.h*头文件将保存在*./includes*中：

hfcal.c需要知道头文件在哪里。

```
gcc -I./includes -c hfcal.c -o hfcal.o
```

记得加-I标志了吗？

-c表示“创建目标文件，但不要链接它”。

2. 接着你需要用测试代码*elliptical.c*创建一个叫*elliptical.o*的目标文件：

```
gcc -I./includes -c elliptical.c -o elliptical.o
```

再强调一遍，你需要告诉编译器头文件在./includes中。

3. 现在你需要用*hfcal.o*创建存档库，并把它保存到*./libs*：

库的名字必须是lib...a。

```
ar -rcs ./libs/libhfcal.a hfcal.o
```

存档需要保存在./libs目录中。

4. 最后，用*elliptical.o*和*hfcal*存档创建`elliptical`可执行文件：

-lhfcal吩咐编译器去找libhfcal.a。

```
gcc elliptical.o -L./libs -lhfcal -o elliptical
```

用elliptical.o和库来构建程序。

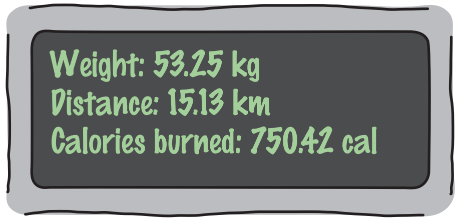
-L./libs告诉编译器库保存在哪里。

你已经建立了elliptical程序，下面就在控制台运行它：

File Edit Window Help SilenceInTheLibrary

```
> ./elliptical
Weight: 115.20 lbs
Distance: 11.30 miles
Calories burned: 1028.39 cal
>
```

事情可没那么简单……



但是有个问题，Head First健身房正在向世界各地扩张，不同国家使用的语言和单位不同。例如在英格兰，器材显示数据的单位是**千克（kg）和千米（km）**。

健身房有多种器材。假如有20种，如果他们要在50个国家开设健身房，那么就需要写1000份不同的软件，这可不是一个小数字。

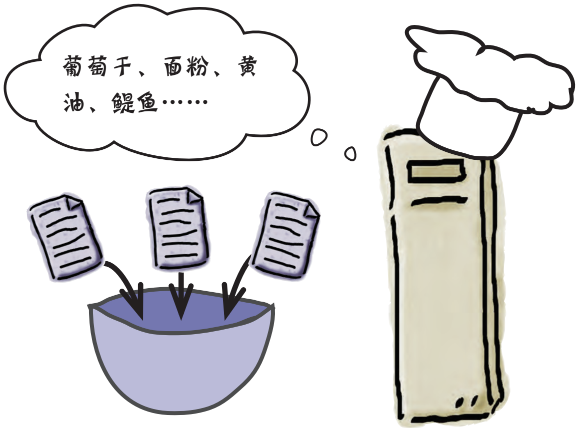
而且还有其他问题：

- 如果工程师升级了某台机器上的传感器，他需要同时升级与传感器交互的代码。
- 如果显示方式改变了，工程师需要修改输出代码。
- 很多其他变化。

仔细想想，你在写其他软件时也会碰到这样的问题。不同的机器需要不同的设备驱动代码，读取不同的数据库，使用不同的图形用户界面。你不可能写出在所有机器上都能运行的代码，这时该怎么办？

程序由碎片组成……

程序是由不同目标代码组建而成的。先创建.*o*文件和.*a*存档，然后再把它们链接成可执行程序。



……一旦链接，就不能改变。

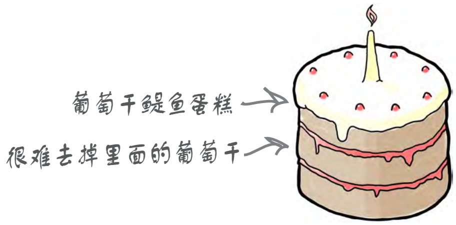
问题是用这种方法构建的程序是**静态**的。一旦用这些独立的目标代码创建了可执行文件，就没有办法修改这些原料，除非重新构建整个程序。

程序链接以后就变成了一大块目标代码。你没有办法把**显示代码**和**传感器代码**分开，它们统统混在了一起。

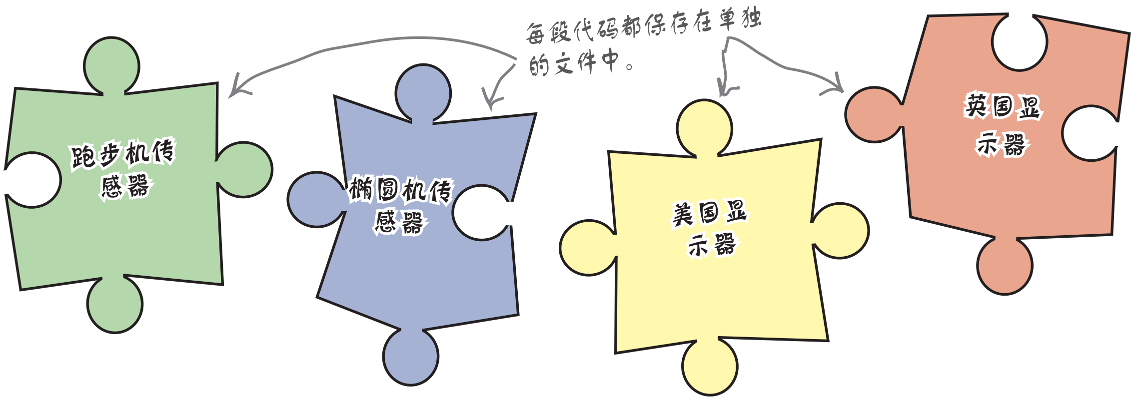
程序要是能使用“热插拔”的目标代码就好了，但我知道我这是在白日做梦……

在运行时动态链接

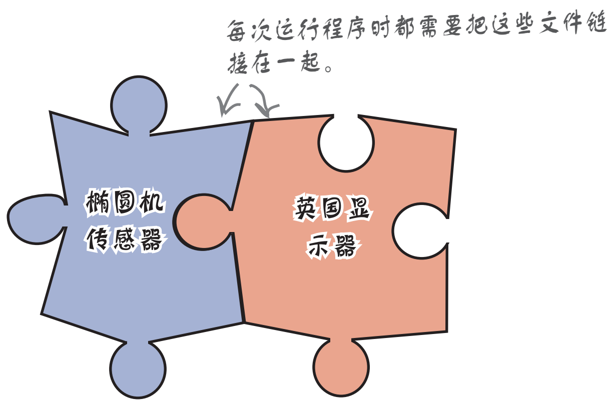
之所以不能修改可执行文件中的目标代码，是因为它们在编译程序时静态链接在了一起。



如果你的程序不是一个文件，而是由很多单独的文件组成，那么在程序运行前把它们链接到一起，就可以避免这个问题。



可以把目标代码分别保存在单独的文件中，在程序运行时才把它们动态链接到一起。



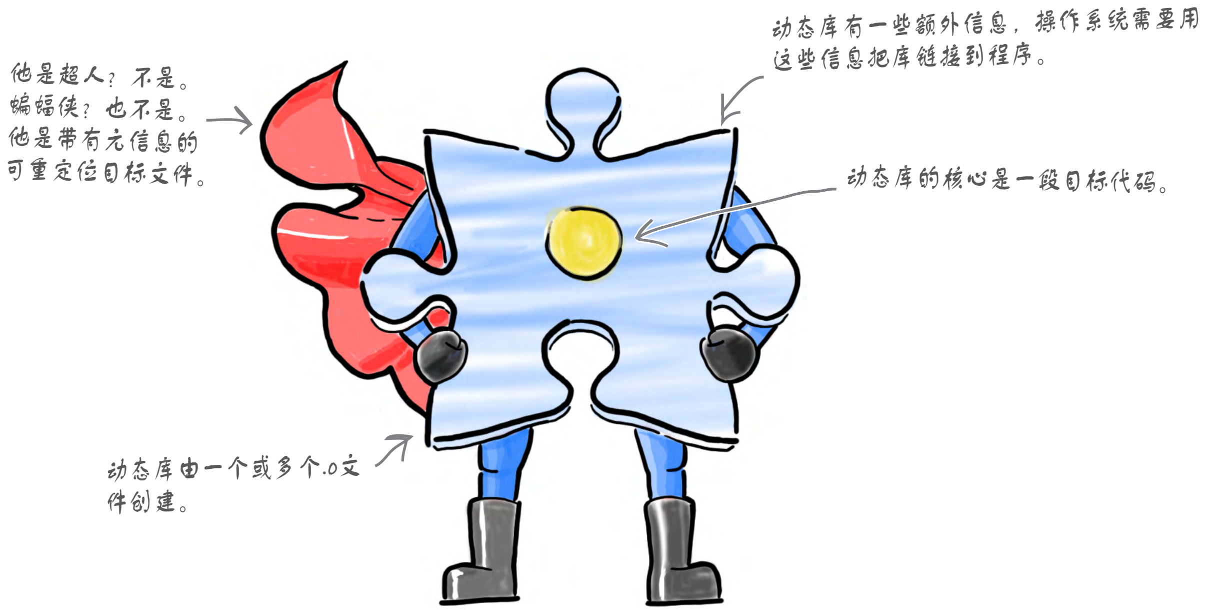
.a能在运行时链接吗？

你需要把目标代码保存在独立的文件中，但*.o*目标文件和*.a*存档文件本身就是独立文件，让计算机在运行程序时链接*.o*文件不就行了么？

事情没有你想象的那么简单。普通的目标文件和存档包含的这点信息还不足以让它们在运行时链接，动态库文件还需要其他东西，例如要链接的文件名。

动态库——加强版目标文件

动态库和你屡屡创建的*.o*目标文件很像，但又不完全一样。动态库和存档也很像，也可以从多个*.o*目标文件创建。不同的是，这些目标文件在动态库中链接成了一段目标代码。



下面就来看看如何创建属于你自己的动态库。

首先，创建目标文件

在把*hfcal.c*代码转换为动态库之前需要把它先编译为*.o*目标文件，像这样：

-c表示“不要链接代码”。

```
gcc -I/includes -fPIC -c hfcal.c -o hfcal.o
```

hfcal.h头文件在/includes中。

-fPIC是什么意思？

发现区别了吗？这次在创建*hfcal.o*时多加了一个标志：**-fPIC**。它告诉gcc你想创建位置无关代码。有的操作系统和处理器要用位置无关代码创建库，这样它们才能在运行时决定把代码加载到存储器的哪个位置。

事实上在大多数操作系统中都不需要加这个选择。试试吧，不加也没有关系。

位置无关代码可以在存储器中搬来搬去。

动手吧！

百宝箱

什么是**位置无关代码**？

位置无关代码就是无论计算机把它加载到存储器的哪个位置都可以运行的代码。想象你有一个动态库，它要使用加载点500个字节以外的某个全局变量的值，那么如果操作系统把库加载到其他地方就会出错。只要让编译器创建位置无关的代码，就可以避免这种问题。

包括Windows在内的一些操作系统在加载动态库时会使用一种叫**存储器映射**的技术，也就是说所有代码其实都是位置无关的。若你在Windows上用刚刚那条命令编译代码，gcc可能会给出一条警告，告诉你不需要-fPIC选项。你既可以奉命删除它，也可以当作没看见。

一种平台一个叫法

绝大部分操作系统都支持动态库，它们的工作方式也大抵相同，但称呼却大相径庭。在Windows中，动态库通常叫动态链接库，后缀名是*.dll*；在Linux和Unix上，它们叫共享目标文件，后缀名*.so*；而在Mac上，它们就叫动态库，后缀名*.dylib*。尽管后缀名不同，但创建它们的方法相同：

```
gcc -shared hfcal.o -o  { C:\libs\hfcal.dll
                        { /libs/libhfcal.dll.a
                        { /libs/libhfcal.so
                        { /libs/libhfcal.dylib
```

C:\libs\hfcal.dll ← Windows上的MinGW

/libs/libhfcal.dll.a ← Windows上的Cygwin

/libs/libhfcal.so ← Linux或Unix

/libs/libhfcal.dylib ← Mac

`-shared`选项告诉gcc你想把*.o*目标文件转化为动态库。编译器创建动态库时会把库的名字保存在文件中，假设你在Linux中创建了一个叫*libhfcal.so*的库，那么*libhfcal.so*文件就会记住它的库名叫*hfcal*。也就是说，一旦你用某个名字编译了库，就不能再修改文件名了，这一点很重要。

若想重命名库，就必须用新的名字重新编译一次。

在一些古老的Mac系统上，没有-shared选项。

别担心，在这些机器上可以用`-dynamiclib`代替。

编译elliptical程序

一旦创建了动态库，你就可以像静态库那样使用它。可以像这样建立`elliptical`程序：

```
gcc -I/include -c elliptical.c -o elliptical.o
gcc elliptical.o -L/libs -lhfcal -o elliptical
```

尽管你使用的命令和静态存档一模一样，但两者编译的方式不同。因为库是动态的，所以编译器不会在可执行文件中包含库代码，而是插入一段用来查找库的“占位符”代码，并在运行时链接库。

下面来看看程序能否运行。

MinGW和Cygwin的库名

在MinGW和Cygwin上，库名的格式有很多种，*hfcal*的库名可以是：

libhfcal.dll.a

libhfcal.dll

hfcal.dll

试驾

你已经在/*libs*目录下创建了动态库，并建立了elliptical测试程序，现在就来运行一下。*hfcal*不在标准目录中，要确保计算机在运行程序时能找到它。

Mac

你可以直接运行程序。当你在Mac中编译程序时，文件的完整路径/*libs*/*libhfcal.dylib*保存在可执行文件中，程序启动时知道去哪里找它。

```
File  Edit  Window  Help  I'mAMac
> ./elliptical
Weight: 115.20 lbs
Distance: 11.30 miles
Calories burned: 1028.39 cal
>
```

← Mac

Linux

但Linux就不一样了。

在Linux和大部分Unix中，编译器只会记录*libhfcal.so*库的文件名，而不会包含路径名。也就是说如果不把*hfcal*库保存到标准目录（如/usr/lib），程序就找不到它。为了解决这个问题，Linux会检查保存在`LD_LIBRARY_PATH`变量中的附加目录。只要把库目录添加到`LD_LIBRARY_PATH`中，并export[①]它，`elliptical`就能找到*libhfcal.so*。

要确保export这个变量。

在Linux上，你需要设置LD_LIBRARY_PATH变量，这样程序才能发现动态库。→

如果动态库已经在标准目录中（如/usr/lib），就不需要这样做。

```
File  Edit  Window  Help  I'mLinux
> export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/libs
> ./elliptical
Weight: 115.20 lbs
Distance: 11.30 miles
Calories burned: 1028.39 cal
>
```

↑ Linux

① export是一条Linux命令，用来将自定义变量设为环境变量。——译者注

Windows

那用Cygwin和MinGW版gcc编译的代码呢？两种编译器都会创建Windows下的DLL库与可执行文件。同Linux一样，Windows可执行文件也只保存*hfcal*库的名字，不保存目录名。

不过Windows没有用LD_LIBRARY_PATH变量去找*hfcal*库。Windows程序会先在当前目录下查找，如果没找到就去查找保存在PATH变量中的目录。

Cygwin

如果用Cygwin编译了程序，可以在Bash shell中这样运行它：

```
File Edit Window Help I'mCygwin
> PATH="$PATH:/libs"
> ./elliptical
Weight: 115.20 lbs
Distance: 11.30 miles
Calories burned: 1028.39 cal
>
```

在Windows中使用Cygwin

MinGW

如果用MinGW编译了程序，可以在命令提示符中这样运行它：

```
File Edit Window Help I'mMinGW
C:\code> PATH="%PATH%:C:\libs"
C:\code> ./elliptical
Weight: 115.20 lbs
Distance: 11.30 miles
Calories burned: 1028.39 cal
C:\code>
```

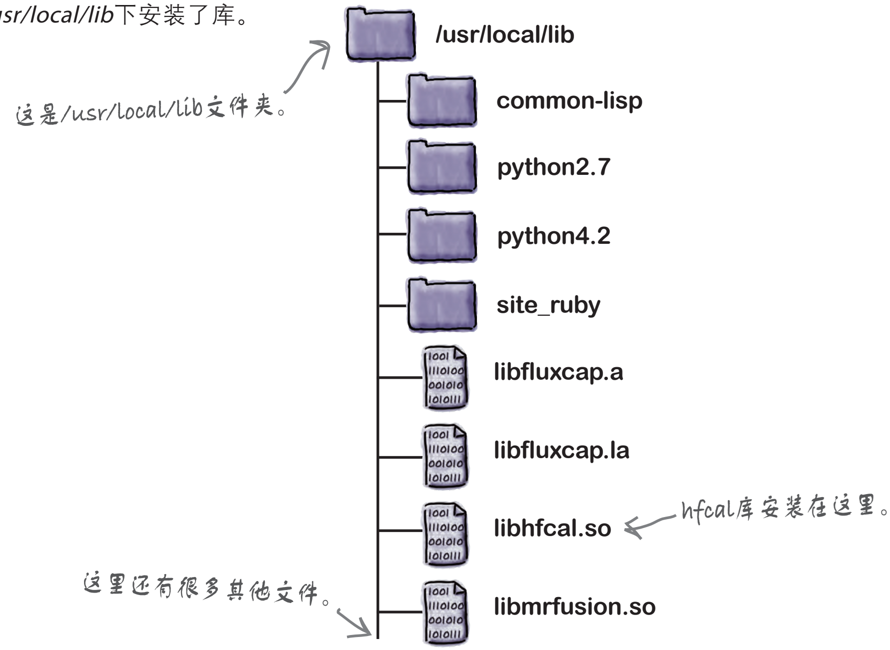
在Windows中使用MinGW

是不是有点复杂？是的，这就是为什么绝大部分使用动态库的程序要把动态库保存在标准目录下。在Linux和Mac中，动态库通常保存在*/usr/lib*或*/usr/local/lib*中；而在Windows中，程序员通常把*.DLL*和可执行文件保存在同一个目录下。

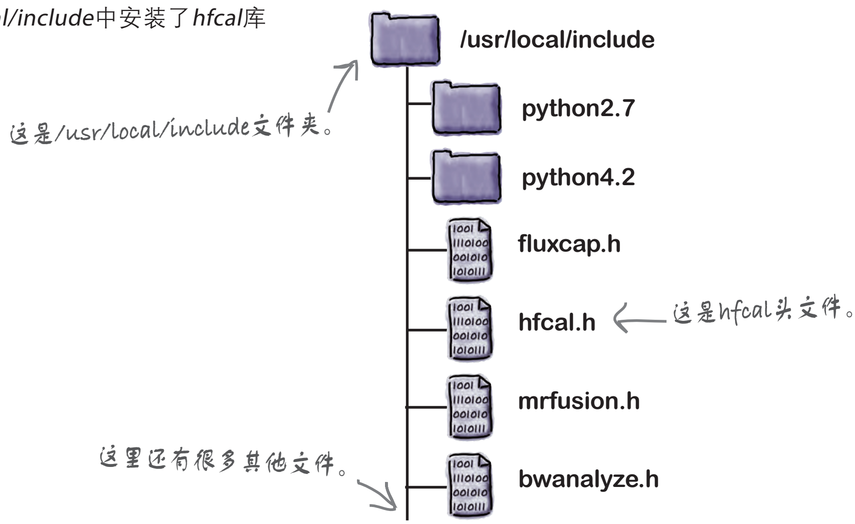
练习

Head First健身房的人正打算把跑步机运输到英格兰。跑步机的嵌入式服务器上装的是Linux，而且已经预装了美版程序。

技术人员在*/usr/local/lib*下安装了库。



在*/usr/local/include*中安装了*hfcal*库的头文件：



技术人员喜欢把库安装在这些目录下，因为它们更“标准”。机器是根据美国人的使用习惯配置的，因此有一些地方需要修改。

为了能在英格兰使用，系统需要进行一些修改：把英里（mile）和磅（pound）换成千米（km）和千克（kg）。

这是在英国健身房中使用的代码。

```
#include <stdio.h>
#include <hfcal.h>

void display_calories(float weight, float distance, float coeff)
{
  printf("Weight: %3.2f kg\n", weight / 2.2046);
  printf("Distance: %3.2f km\n", distance * 1.609344);
  printf("Calories burned: %4.2f cal\n", coeff * weight * distance);
}
```

代码以km与kg为单位显示信息。

hfcal_UK.c

这个文件在/home/ebrown目录下。

安装在机器上的软件需要使用这段新代码。因为软件会以动态库的形式链接这段代码，所以只要把代码编译到*/usr/local/lib*目录下就行了。

假设你已经进入了*hfcal_UK.c*文件的目录，有所有目录的写权限。为了编译新版动态库，你需要输入什么命令？

....

....

....

假设跑步机的主程序叫*/opt/apps/treadmill*，为了运行程序，需要输入什么命令？

....

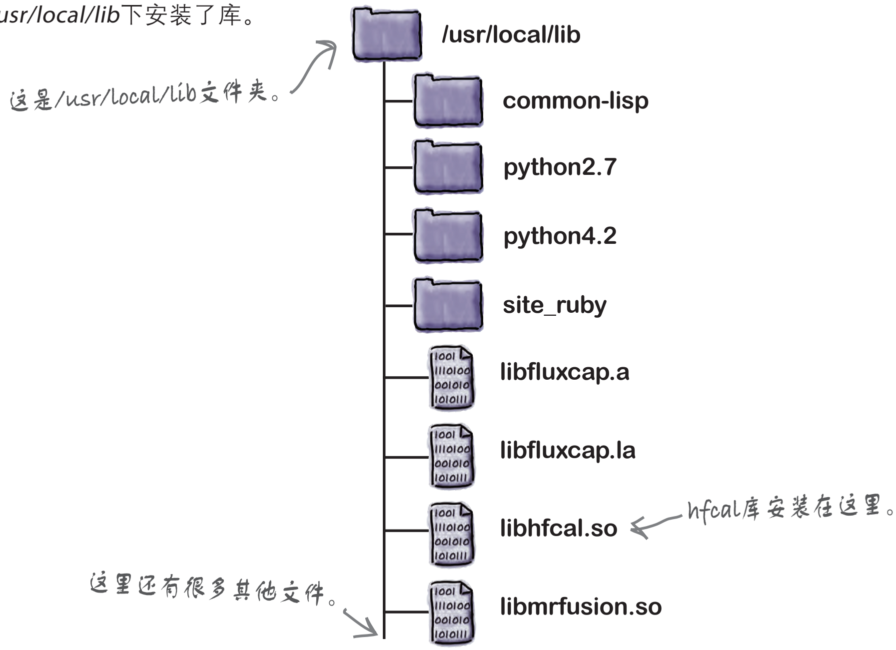
....

....

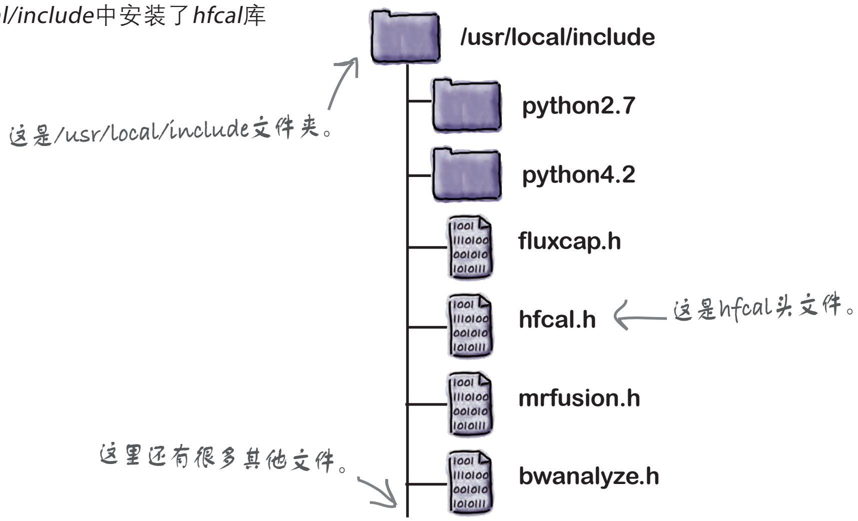
练习解答

Head First健身房的人正打算把跑步机运输到英格兰。跑步机的嵌入式服务器上装的是Linux，而且已经预装了美版程序。

技术人员在*/usr/local/lib*下安装了库。



在*/usr/local/include*中安装了*hfcal*库的头文件：



技术人员喜欢把库安装在这些目录下，因为它们更“标准”。机器是根据美国人的使用习惯配置的，因此有一些地方需要修改。

为了能在英格兰使用，系统需要进行一些修改：把英里（mile）和磅（pound）换成千米（km）和千克（kg）。

```
#include <stdio.h>
#include <hfcal.h>

void display_calories(float weight, float distance, float coeff)
{
  printf("Weight: %3.2f kg\n", weight / 2.2046);
  printf("Distance: %3.2f km\n", distance * 1.609344);
  printf("Calories burned: %4.2f cal\n", coeff * weight * distance);
}
```

hfcal_UK.c

安装在机器上的软件需要使用这段新代码。因为软件会以动态库的形式链接这段代码，所以只要把代码编译到*/usr/local/lib*目录下就行了。

假设你已经进入了*hfcal_UK.c*文件的目录，有所有目录的写权限。为了编译新版动态库，你需要输入什么命令？

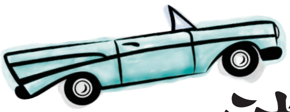
需要把源代码编译为目标文件。→ gcc -c -fPIC hfcal_UK.c -o hfcal.o ← 不用设置-I选项，因为头文件在标准目录中。

然后需要把目标文件转化为共享目标文件。→ gcc -shared hfcal.o -o /usr/local/lib/libhfcal.so

假设跑步机的主程序叫*/opt/apps/treadmill*，为了运行程序，需要输入什么命令？

/opt/apps/treadmill ← 不用设置LD_LIBRARY_PATH变量，因为动态库在标准目录中。

你发现了吗？动态库和头文件已经安装在了标准目录中，所以在编译代码时不需要使用-I标志，运行代码时也不需要设置LD_LIBRARY_PATH变量。

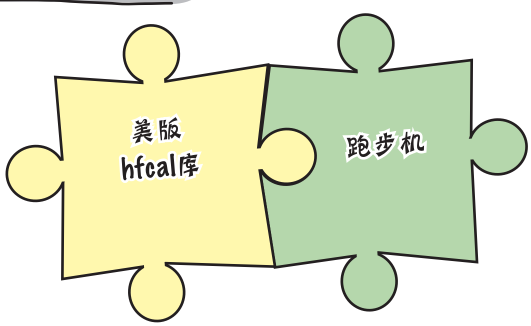


试驾

你已经修改了英版跑步机上的代码，我们对比一下**美版**跑步机。下面这台美版跑步机使用了原版的*libhfcal.so*库。

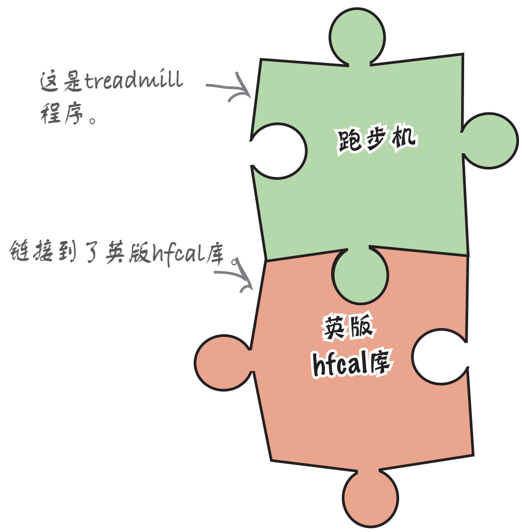
机器启动时运行了`treadmill`程序，当用户在跑步机上跑了一段时间以后，显示如下：

美版跑步机上的`treadmill`程序动态链接到了*libhfcal.so*库，而*libhfcal.so*是用美版*hfcal*程序编译的。

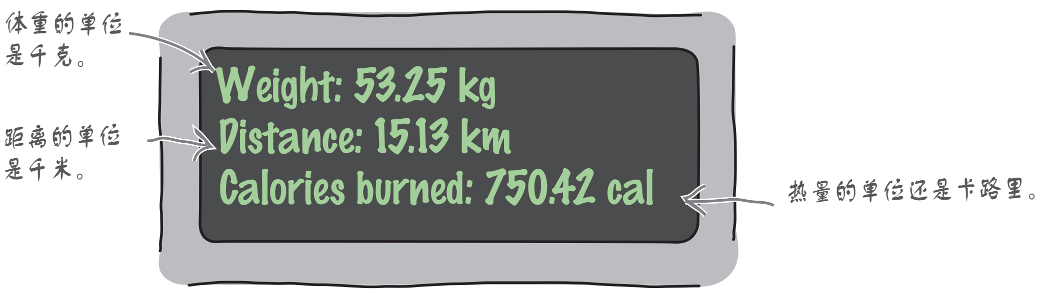


英版的呢?

英版跑步机安装了相同的`treadmill`程序，但你用*hfcal_UK.c*文件中的源代码重新编译了*libhfcal.so*库。



当用户跑了一段相同的距离以后，显示如下：



正确运行了。
`treadmill`程序不需要重新编译就能从新的库中**动态**获取代码。

有了动态库，就能在**运行时**替换代码。不用重新编译程序，你就能修改它。如果你有很多程序，它们共享一段相同的代码，通过建立动态库，就可以同时更新所有程序。既然你已经学会了创建动态库，也就成为了一名更厉害的C程序员。

今日主题：两位著名软件模块化粉丝正在讨论静态链接与动态链接的利弊。

静态： 我们都赞成写模块化程序。

动态： 当然。

静态： 顺理成章的一件事。

动态： 不错。

静态： 让代码易于管理。

动态： 对。

静态： 这样就能写大程序了。

动态： 大程序？

静态： 嗯，把所有要用到的东西都放进一个可执行文件。

动态： 这可不是什么好主意。

静态： 何出此言？老朋友。

动态： 我认为程序应该由多个小文件链接而成，只有在运行时才链接它们。

静态： 啊？（大笑）……你不是在开玩笑吧？

动态： 我是认真的。

静态： 啊？多个独立文件？仓促地链接在一起？！

动态： 不是仓促，是动态。

静态： 这是混乱的源头！

动态： 这样做我就能在之后改变主意。

静态： 你应该一次性就把事情做对。

动态： 不可能一次性就做对，所有大程序都应该动态链接。

静态： 所有程序？

动态： 是的。

静态： 那Linux内核怎么算？够大了吧？它可是……

动态： ……静态链接的，我知道。你赢了一次。

静态： 静态链接的机动性不强，但用起来很简单，一个文件打天下，如果你想安装程序，只要拷贝可执行文件即可，不需要DLL之类的鬼东西。

动态： 我们谁也说服不了谁。

静态： 看样子我不能改变你的想法。

动态： 是的。

静态： 所以你也是静态链接的。

要点

- 动态库在运行时链接程序。
- 用一个或多个目标文件创建动态库。
- 在一些机器上，需要用`-fPIC`选项来编译目标文件。
- `-fPIC`令目标代码位置无关。
- 在一些机器上，可以省略`-fPIC`。
- `-shared`编译选项可以创建动态库。
- 动态库在不同机器上名字不同。
- 如果把动态库保存在标准目录中，生活会变得更简单。
- 不然，就需要设置`PATH`变量和`LD_LIBRARY_PATH`变量。

这里没有蠢问题

问：为什么动态库在不同操作系统中如此不同？

答： 操作系统喜欢优化加载动态库的方式，因此不同操作系统对动态库制定了不同的需求。

问：我想改变动态库的名字，于是重命名了文件名，但编译器找不到它，为什么？

答： 编译器在编译动态库时会在文件中保存库名。如果你重命名了文件，文件中的名字还是没变。如果想修改动态库的名字就必须重新编译它。

问：为什么Cygwin的动态库文件支持多种不同的命名方式？

答： 因为Cygwin专门用来在Windows上编译Unix软件。Cygwin会创建一个类似Unix的环境，因此借鉴了很多Unix的命名约定。比如用*.a*来命名库，即使它们是动态DLL。

问：Cygwin动态库是真正的DLL吗？

答： 是的，但因为它们是基于Cygwin的，如果你想在一般的Windows程序中使用Cygwin动态库，还需要做一些工作。

问：为什么MinGW动态库的命名格式和Cygwin一样？

答： 这两个项目的关系十分紧密，而且共享了很多代码。它们最大的区别是用MinGW编译的程序在没有安装Cygwin的计算机上也能够运行。

问：为什么Linux不直接在可执行文件中保存库路径名？那样不就可以不用设置`LD_LIBRARY_PATH`了吗？

答： 这是一种设计上的选择，如果不保存路径名，程序就可以使用不同版本的库。当你要开发新的库时，这种设计的好处就特别明显。

问：为什么Cygwin不用`LD_LIBRARY_PATH`来查找库？

答： 因为Cygwin使用Windows的DLL，Windows会用`PATH`变量来加载DLL。

问：静态链接和动态链接哪个好？

答： 不可一概而论。使用静态链接，可以得到一个小而快的可执行文件，并可以很方便地把它从一台机器拷贝到另一台。而动态链接允许在运行时配置程序。

问：如果不同的程序使用相同的动态库，动态库会加载一次还是多次？这些程序会共享它吗？

答： 这取决于操作系统。有的操作系统会为每个进程加载一个动态库，有的则会共享动态库以节省存储器。

问：动态库是配置程序的最好方式吗？

答： 通常情况下，配置文件可能比动态库更简单。但如果想连接一些外部设备，通常会用动态库作为驱动。

C语言工具箱

你已经学完了第8章，现在你的工具箱又加入了静态库和动态库。关于本书的提示工具条的完整列表，请见附录ii。

-L<路径名>在标准lib目录列表中添加目录。

#include<>会查找包括/usr/include在内的标准目录。

-l<路径名>会链接标准目录（例如/usr/lib）下的文件。

-I<路径名>在标准include目录列表中添加目录。

ar命令创建目标文件的存档。

库存档名形如libXXX.a。

gcc -shared把目标文件转化为动态库。

库存档是静态链接的。

动态库在运行时链接。

动态库在不同的操作系统上有不同的名字。

动态库的后缀名有.so、.dylib、.dll和.dll.a等。

实验人：

实验日期：

C语言实验室 2

OpenCV

本实验会给你一份说明书，它描述了一个程序，你需要运用你在前几章中学到的知识构建这个程序。

这个项目比你之前见识到的项目都要大，所以动手之前请阅读完全部内容，并给自己一点时间。不要担心会被难倒，这里没有新概念，你也可以接着往后读，回过头再来做这个实验。

该去完成任务了，我们不会提供任何代码或答案。

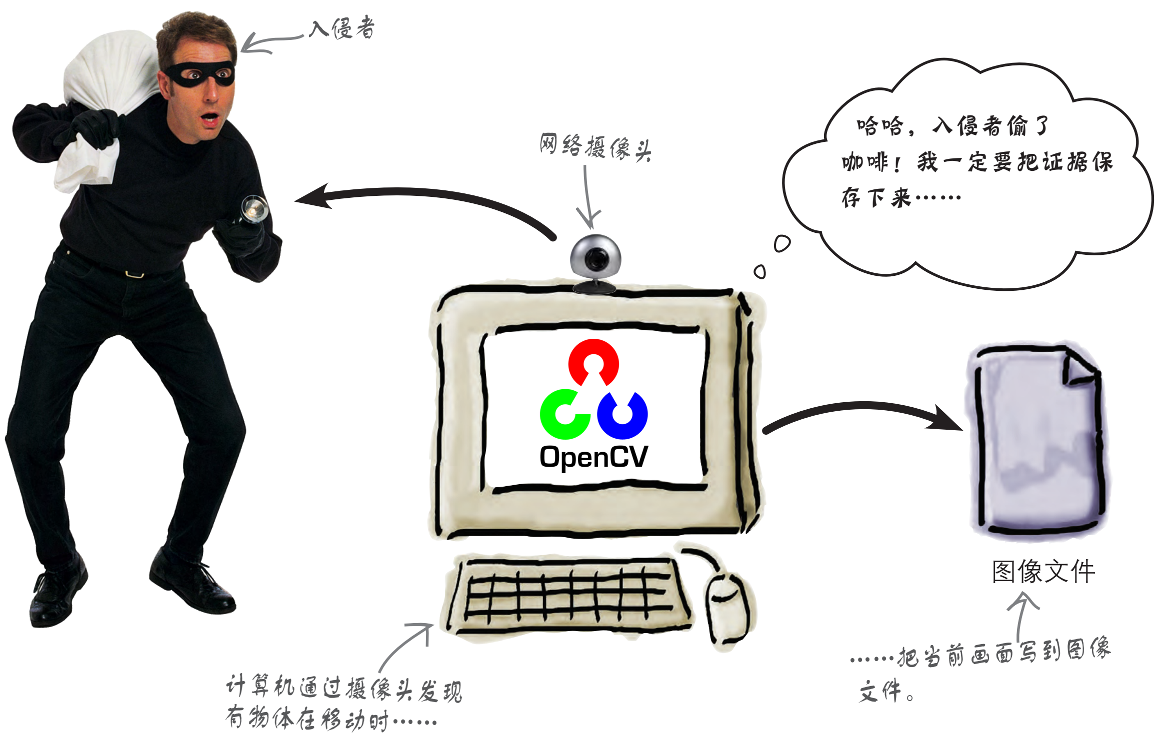
说明书：入侵者检测器

试想一下，当你出门在外，如果你的计算机能帮你看家，还能让你看到小偷的真面目，该是多么神奇的一件事！这不是在做梦，只要计算机有网络摄像头，加上OpenCV的神奇力量就能做到！

你将创建：

入侵者检测器

计算机会用网络摄像头持续监测周围环境，当检测到有物体在移动时就会把当前捕捉到的图像保存为文件。如果把这个文件保存在网络驱动器上，或使用Dropbox那样的文件同步服务，就能抓他个“正着”。



OpenCV

OpenCV是一款开源计算机视觉库，可以用它获取摄像头的输入、处理图像、分析实时图像数据，并根据计算机看到的东西判断有没有小偷。最重要的是，这一切都可以通过C代码来实现。

你可以在Windows、Linux和Mac平台上使用OpenCV，通过以下链接访问OpenCV的wiki页面：

http://opencv.willowgarage.com/wiki/FullOpenCVWiki

安装OpenCV

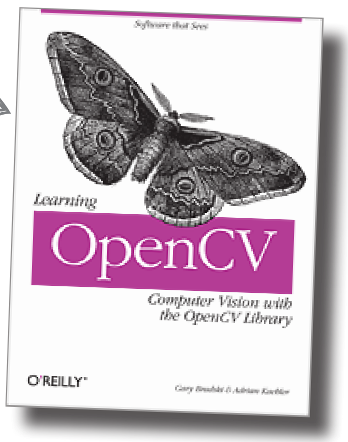
你可以在Windows、Linux或Mac中安装OpenCV，下面是安装指南，里面包含了OpenCV最新稳定版的下载链接：

http://opencv.willowgarage.com/wiki/InstallGuide

安装完以后，你会在计算机上找到一个叫*samples*的文件夹，打开瞧瞧，里面有一些OpenCV的wiki链接。为了完成实验，你应该调研一番。

如果想深入了解OpenCV，我们推荐Gary Bradski和Adrian Kaehler的《学习OpenCV》 。

这本书能让你在学习OpenCV时如饮醍醐。



代码应完成

你的C代码应该完成：

获取输入

你要处理摄像头拍摄的实时数据，因此你要做的第一件事就是捕获这些数据。有个叫**cvCreateCameraCapture(0)**的OpenCV函数可以帮到你。它返回一个指向CvCapture结构的指针，通过这个指针你就可以访问摄像头设备并获取图像。

计算机有可能找不到摄像头，所以调用函数时别忘了检查错误。如果无法访问摄像头，cvCreateCameraCapture(0)会返回NULL指针。

图像文件

捕获图像

你可以用cvQueryFrame()函数读取摄像头拍到的最新图像。它接收CvCapture指针作为参数，返回一个指向最新图像的指针。代码在开始时可能看起来像这样：

```
CvCapture* webcam = cvCreateCameraCapture(0);
if (!webcam)
  /* 退出并置错误码 */
while (1) {
  IplImage* image = cvQueryFrame(webcam);
  if (image) {

  }
}
```

- if (!webcam) ← 说明“找不到摄像头”。
- while (1) { ← 无限循环。
- 从网络摄像头读取图像。→ IplImage* image = cvQueryFrame(webcam);
- 如果读取到图片，就在这里处理。

只要能肯定这幅图像中有小偷，就可以用下面这行代码把图像保存为文件：

```
cvSaveImage("somefile.jpg", image, 0);
```

- "somefile.jpg"：图像文件名。
- image：从摄像头读取到的图像。
- 0：如果不想保存为灰度图就置0。

检测入侵者

接下来是代码中最巧妙的部分：如何判断某一帧图像中出现了入侵者。

有一种方法是检测图像的移动量。OpenCV提供了一些创建Farneback光流的函数。光流会比较两幅图像，然后告诉你像素移动了多少距离。

这部分内容需要你自己研究，你有可能会用`cvCalcOpticalFlowFarneback()`来比较两幅连续的图像，并创建光流。所以你需要写一些代码来测量两帧画面之间的移动量。一旦移动量超过了某个阈值，你就知道有个大家伙在摄像头前移动。

全身而退

当启动程序时，你可不希望摄像头把你走开的这个过程也记录下来，因此需要添加一段延时，好让你有时间离开房间。

可选：显示当前画面

测试期间，我们希望能看到当前程序“看到”的那帧画面，为此我们打开一个窗口，用它显示当前网络摄像头的输出。

只要用以下命令就可以在OpenCV中创建窗口：

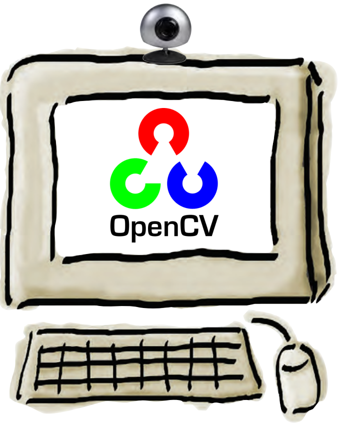
```
cvNamedWindow("Thief", 1);
```

在窗口中显示当前图像：

```
cvShowImage("Thief", image);
```

检测器下线

当计算机能自动拍下那些鬼鬼祟祟的家伙，就说明你的OpenCV项目已经完成了。



为什么不接着往下讲？因为我们确信你可以用OpenCV做出更多意想不到的事情。欢迎给Head First实验室写信，让我们知道你使用OpenCV的情况。

勇者之路

本书最后会介绍一些**高级主题**。

在你开始探索C语言的高级功能前，请确保你的计算机能够使用这些特性。如果你用的是Linux或Mac，很好，但如果你用的是Windows，需要先安装Cygwin。

准备好了的话就翻到下一页，挺胸走进大门……

9 进程与系统调用

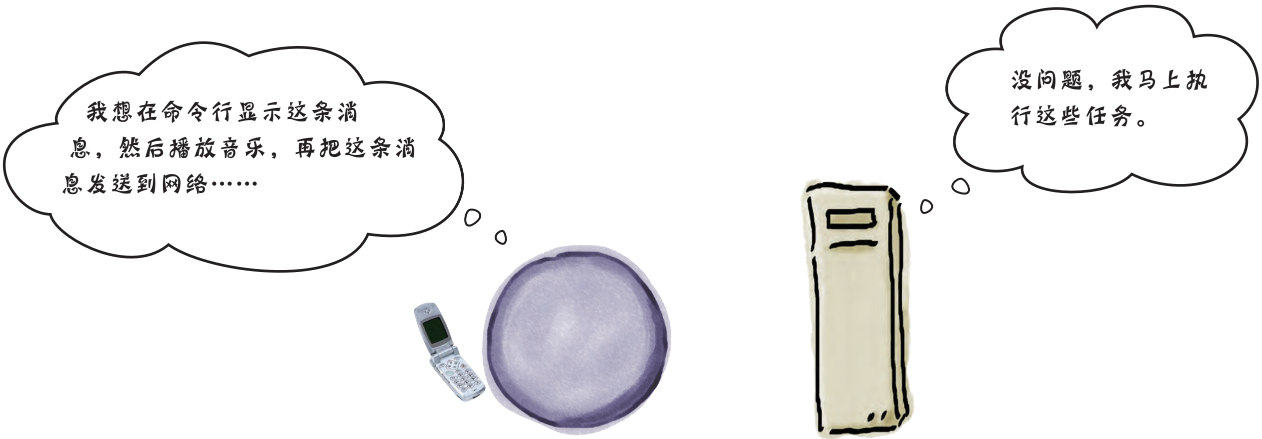
打破疆界

打破常规。

你已经学会了通过在命令行连接小工具的方式建立复杂的程序。但如果你想在代码中使用其他程序怎么办？本章中你将学会如何用系统服务来创建和控制进程，让程序发电子邮件、上网和使用任何已经安装过的程序。本章的最后，你将得到超越C语言的力量。

操作系统热线电话

C程序无论做什么事都要靠操作系统。如果它想与硬件打交道，就要进行**系统调用**。系统调用是操作系统**内核**中的函数，C标准库中大部分代码都依赖于它们。每当调用`printf()`在命令行显示字符串时，C程序都会在幕后向操作系统发出系统调用，把字符串发送到屏幕。



下面来看一个系统调用的例子，我们将从一个名副其实的系统调用——**system()**开始。

`system()`接收一个字符串参数，并把它当成命令执行：

```
system("dir D:");
system("gedit");
system("say 'End of line'");
```

- `system("dir D:");` ← 打印D盘内容。
- `system("gedit");` ← 在Linux中启动编辑器。
- `system("say 'End of line'");` ← 在Mac上朗读文本。

`system()`函数是在代码中运行其他程序的捷径，特别是在建立快速原型时，与其写很多C代码，不如调用外部程序。

代码冰箱贴

下面这个程序将一段带有时间戳的文本写到日志文件的底部。整个程序都可以用C语言来写，但程序员用了system()调用，因为它可以更快速地处理文件。

你能补全代码吗？代码创建了一条命令字符串，它先显示注释文本，接着是时间戳。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

char* now()
{
  time_t t;
  time (&t);
  return asctime(localtime (&t));
}

/* 主控程序，用来登记警卫的巡逻记录。 */
int main()
{
  char comment[80];
  char cmd[120];

  ..........(..........,..........,..........);

  ..................(....................,

        ....................,

        ..................,..................);
  system(cmd);
  return 0;
}
```

函数返回一个字符串，包含当前文本和时间。

sprintf

"echo '%s %s' >> reports.log"

80

stdin

cmd

printf

120

comment

fgets

comment

now()

scanf

stdout

代码冰箱贴解答

下面这个程序将一段带有时间戳的文本写到日志文件的底部。整个程序都可以用C语言来写，但程序员用了system()调用，因为它可以更快速地处理文件。

你将补全代码。代码创建了一条命令字符串，它先显示注释文本，接着是时间戳。

```
#include <stdio.h>
#include <stdlib.h>
#include <time.h>

char* now()
{
  time_t t;
  time (&t);
  return asctime(localtime (&t));
}

/* 主控程序，用来登记警卫的巡逻记录。 */
int main()
{
  char comment[80];
  char cmd[120];
  fgets(comment, 80, stdin);
  sprintf(cmd, "echo '%s %s' >> reports.log",
    comment, now());
  system(cmd);
  return 0;
}
```

需要把文本保存在comment数组中。

只有80个字符的空间。

从标准输入（也就是键盘）读取数据。

用fgets读取非结构化文本。

sprintf会把这些字符写到字符串中。

格式化字符串将保存在cmd数组中。

命令模板。

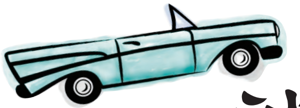
命令会把注释追加到文件底部。

注释先出现。

时间戳后出现。

运行cmd字符串的内容。

scanf　stdout　120　printf



试驾

编译程序，看看它是如何工作的：

编译程序。
运行程序。
再运行一次

```
File Edit Window Help Who'sYourUser
> gcc guard_log.c -o guard_log
> ./guard_log
Checked in Crom - a compound interest program.
> ./guard_log
Blue Leader reports breach in jet walls.
>
```

注释。
另一条注释。

当你查看程序所在目录时，程序创建了一个叫*reports.log*的新文件。

```
Checked in Crom - a compound interest program.
 Thu Oct 29 11:25:53 2015

Blue Leader reports breach in jet walls.
 Thu Oct 29 11:26:06 2015
```

这些是时间戳。

这是程序创建的reports.log文件。

reports.log

程序工作了。它从命令行读取注释，然后调用echo命令把注释追加到文件底部。

整个程序都可以用C语言来写，但你用system()简化了程序，可谓事半功倍。

这里没有蠢问题

问： system()函数会编译到我的程序中吗？

答： 不会，和所有系统调用一样，system()函数不在你的程序里，而在操作系统中。

问： 所以我在进行系统调用时会调用外部代码，像库一样，是吗？

答： 差不多，但具体细节要看操作系统。在一些操作系统中，系统调用的代码位于操作系统内核。而对其他操作系统而言，系统调用可能保存在动态库中。

黑客入侵了……

system()函数也有不好的一面。虽然它用起来很简单，上手很快，但也疏忽了很多东西。在正视这些问题以前，我们先来看看如何入侵程序。

代码通过拼接命令字符串的方式工作，像这样：

```
echo ' | <comment> | | <timestamp> | ' >> reports.log
```

但如果有人输入了这样的命令怎么办？

```
echo ' | ' && ls / && echo ' | | <timestamp> | ' >> reports.log
```

通过在文本中注入命令行代码，就能随心所欲地让程序运行任何命令：

用户可以用这个程序随心所欲地在计算机上运行任何命令。

```
File Edit Window Help Yikes
> ./guard_log
' && ls / && echo '

Applications                System   dev           private
Developer                   Users    etc           sbin
Library                     Volumes  home          tmp
Network                     bin      mach_kernel   usr
Space Paranoids Source      cores    net           var
>
```

列出了根目录下的内容。

这个问题很严重吗？只要用户能运行guard_log，就能轻易地运行其他程序，如果你的代码是在服务器上调用的怎么办？如果这是一个处理文件数据的程序怎么办？

岂止是安全问题

刚刚的例子在程序中注入了一段“列出根目录内容”的代码，它也可以删除文件或启动病毒。但你不应该只关注安全问题。

- **注释文本中出现了撇号怎么办？**
  这会破坏echo命令中的引号。
- **PATH变量让system()函数调错了程序怎么办？**
- **需要先设置一批专门的环境变量，程序才能工作，怎么办？**

`system()`函数用起来方便，但很多时候需要更规范的方法。你需要用命令行参数甚至是环境变量调用*指定*程序。

百宝箱

什么是内核？

在大部分计算机上，系统调用就是操作系统内核中的函数。什么是内核？虽然你从来没在屏幕上看到过它，但内核其实一直都在那里控制计算机。内核是计算机中最重要的程序，它主管三样东西：

进程
只有当内核把程序加载到存储器时程序才能运行。内核创建进程，并确保它们得到了所需资源。内核同时也会留意那些变得贪得无厌或者已经崩溃的进程。

存储器
计算机所能提供的存储器资源是有限的，因此内核必须小心翼翼地分配每个进程所能使用的存储器大小。内核还能把部分存储器交换到磁盘从而增加虚拟存储器空间。

硬件
内核利用设备驱动与连接到计算机上的设备交互。你的程序在不了解键盘、屏幕和图形处理器的情况下就能使用它们，因为内核会代表你与它们交涉。

系统调用是程序用来与内核对话的函数。

exec()给你更多控制权

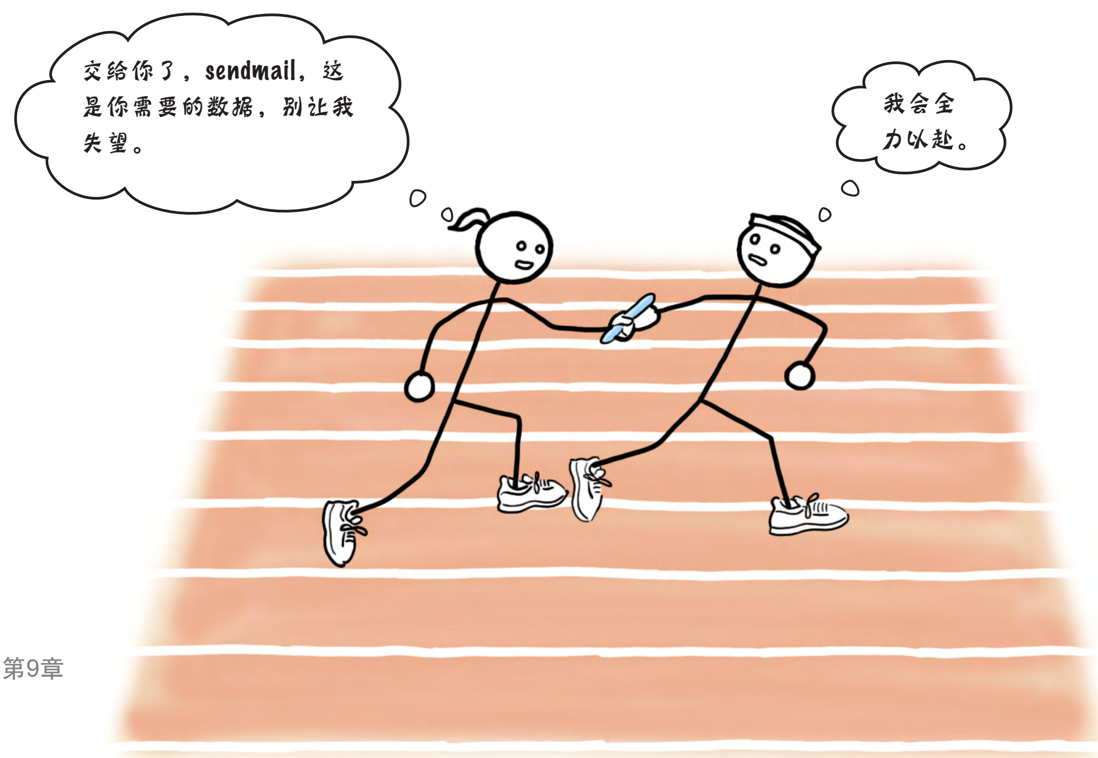
当调用`system()`函数时，操作系统必须解释命令字符串，然后决定运行哪些程序和怎样运行。问题就出在“操作系统需要解释字符串”上，你已经看到这有多么容易出错。要想解决这个问题就必须消除歧义，明确地告诉操作系统你想运行哪个程序，这就是**`exec()`**函数的用处。

exec()函数替换当前进程

进程是存储器中运行的程序。如果在Windows中输入**`taskmgr`**，或在Linux或Mac上面输入**`ps -ef`**，就可以看到系统中运行的进程。操作系统用一个数字来标识进程，它叫**进程标识符**（process identifier，简称PID）。

进程是存储器中运行的程序。

`exec()`函数通过运行其他程序来替换当前进程。你可以告诉`exec()`函数要使用哪些命令行参数和环境变量。新程序启动后PID和老程序一样，就像两个程序接力跑，你的程序把进程交接给了新程序。



exec()函数有很多

久而久之，程序员创建了很多不同版本的exec()。每个版本的名字都有一些细微差别，而且有各自的参数。虽然exec()函数的版本众多，但可以分为两组：**列表**函数和**数组**函数。

exec()函数在unistd.h中。

列表函数：execl()、execlp()、execle()

列表函数以参数列表的形式接收命令行参数：

- **程序。**
  第一个参数告诉exec()函数将运行什么程序。对execl()或execle()来说，它是程序的完整路径名；对execlp()来讲就是命令的名字，execlp()会根据它去查找程序。

- **命令行参数。**
  你需要依次列出想使用的命令行参数。别忘了，**第一个**命令行参数必须是程序名，也就是说列表版exec()的前两个参数是相同字符串。

- **NULL。**
  没错，需要在最后一个命令行参数后加上**NULL**，告诉函数没有其他参数了。

- **环境变量（如果有的话）。**
  如果调用了以...e()结尾的exec()函数，还可以传递环境变量数组，像“POWER=4”、“SPEED=17”、“PORT=OPEN”……那样的字符串数组。

命令行参数之间的空格会把MinGW弄糊涂。

如果把“I like”和“turles”这两个参数传给exec()，MinGW程序可能会发送三个参数：“I”、“like”和“turtle”。

execL = 参数列表(List)。

这些是参数。

```
execl("/home/flynn/clu", "/home/flynn/clu", "paranoids", "contract", NULL)
```

应该用NULL来结束列表。

第二个参数应该和第一个相同。

execLP = 参数列表（List）+ 在PATH中查找程序。

这些是参数。

```
execlp("clu", "clu", "paranoids", "contract", NULL)
```

这些是参数。

```
execle("/home/flynn/clu", "/home/flynn/clu", "paranoids", "contract", NULL, env_vars)
```

execLE = 参数列表（List）+ 环境变量（Environment）。

env_vars是一个字符串数组，里面放了环境变量。

数组函数：execv()、execvp()、execve()

如果已经把命令行参数保存在了数组中，就会发现这两个版本用起来更容易：

execv＝参数数组或参数向量（vector）。

```
execv("/home/flynn/clu", my_args);
```

execvP＝参数数组/向量（vector）＋在PATH中查找。

```
execvp("clu", my_args);
```

参数需要保存在字符串数组my_args中。

上面两个函数的唯一区别就是**execvp**会用PATH变量查找程序。

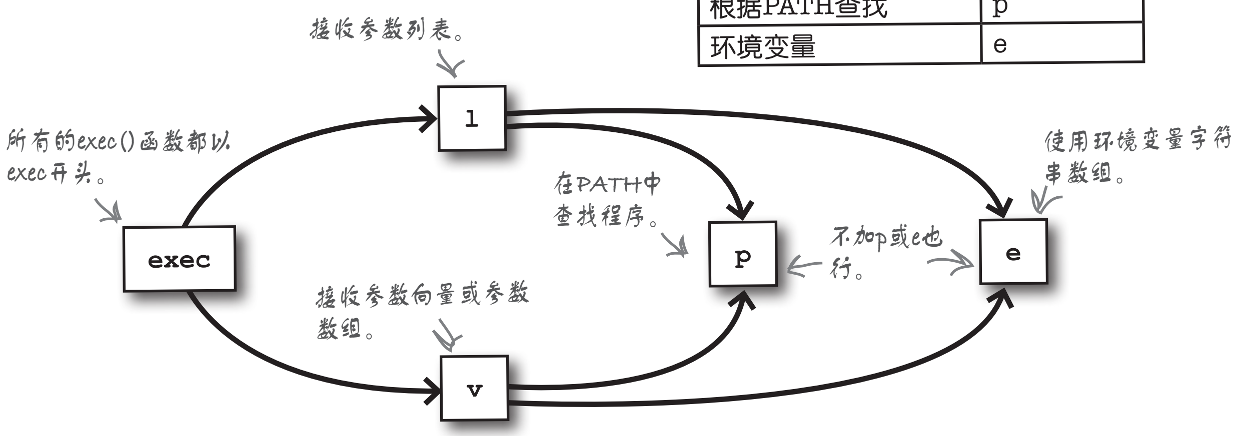
教你如何记住exec()函数

可以通过构造名称的方法来找到你需要的exec()函数。每个exec()函数名之后可以跟一到两个字符，但只能是**l**、**v**、**p**和**e**中的一个。它们分别代表你想使用的功能。对execle()函数来讲：

execle ＝ exec ＋ l ＋ e ＝ 参数列表+环境变量

l、v总是在p、e之前出现；p、e是可选的。

使用	字符
参数列表	l
参数数组/向量	v
根据PATH查找	p
环境变量	e



传递环境变量

每个进程都有一组环境变量。你可以在命令行中输入set或env查看它们的值，它们一般会告诉进程一些有用的信息，比如用户主目录的位置，或去哪里找命令。C程序可以用**getenv()**系统调用读取环境变量，右侧的diner_info程序演示了getenv()的使用方法。

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  printf("Diners: %s\n", argv[1]);
  printf("Juice: %s\n", getenv("JUICE"));
  return 0;
}
```

你可以使用stdlib.h中的getenv()读取环境变量。

diner_info.c

如果你想用命令行参数和环境变量运行程序，可以这样做：

可以用字符串指针数组的形式创建一组环境变量。

环境变量的格式是“变量名=值”。

数组最后一项必须是NULL。

```
char *my_env[] = {"JUICE=peach and apple", NULL};

execle("diner_info", "diner_info", "4", NULL, my_env);
```

execle传递参数列表和环境变量。

my_env里放的是环境变量。

execle()函数将设置命令行参数和环境变量，然后用diner_info替换当前进程。

```
File Edit Window Help MoreOJ
> ./my_exec_program
Diners: 4
Juice: peach and apple
>
```

出错了怎么办？

如果在调用程序时发生错误，当前进程会继续运行。这点很有用，因为就算第二个进程启动失败，还是能够从错误中恢复过来，并向用户报告错误信息。而且幸运的是，C标准库提供了一些内置代码帮你做这些事。

在Cygwin中传递环境变量时一定要包含PATH变量。

在Cygwin中，加载程序时需要用PATH变量，因此在Cygwin上传递环境变量时一定要包含PATH=/usr/bin。

大多数系统调用以相同方式出错

由于系统调用依赖于程序以外的东西，所以它们一旦出错，就没办法控制。为了解决这个问题，系统调用总是以相同方式出错。

就拿execle()调用来说，判断exec()有没有出错很容易：如果exec()调用成功，当前程序就会停止运行。一旦程序运行了exec()以后的代码，就说明出了问题。

失败的标准

如果execle()执行成功，这行代码就不会运行。

```
execle("diner_info", "diner_info", "4", NULL, my_env);
puts("哥们，diner_info程序肯定发生了什么问题");
```

但仅仅告诉用户系统调用失败与否是不够的，通常你想知道系统调用为什么失败，因此几乎所有系统调用都遵循"**失败黄金法则**"。

失败黄金法则

* 尽可能收拾残局。
* 把errno变量设为错误码。
* 返回-1。

errno变量是定义在*errno.h*中的全局变量，和它定义在一起的还有很多标准错误码，如：

EPERM=1	不允许操作
ENOENT=2	没有该文件或目录
ESRCH=3	没有该进程
EMULLET=81	发型很难看

这个值在任何系统上都不存在。

这样你就可以拿**errno**和这些值比较，也可以用*string.h*中的strerror()的函数查询标准错误消息：

```
puts(strerror(errno));
```

strerror()将错误码转换为一条消息。

当系统找不到你想运行的程序时就会把errno变量设置为ENOENT，以上代码就会显示这条消息：

```
没有该文件或目录
```

你可以在不同的机器上用不同命令查看网络配置。在Linux和Mac上，你可以用一个叫/sbin/ifconfig的程序；而在Windows上可以用ipconfig的命令，它的路径保存在命令路径中。

下面这个程序试图运行/sbin/ifconfig程序，如果失败就运行ipconfig命令。你不用传递任何参数给这两条命令，仔细考虑需要使用什么类型的exec()命令？

```
#include <stdio.h>
..................................................
..................................................
..................................................

int main()
{
  if (.................................................................................)
    if (execlp(..........................................................................) {
      fprintf(stderr, "Cannot run ipconfig: %s", ..................................);
      return 1;
    }
  return 0;
}
```

需要哪些头文件？

需要运行/sbin/ifconfig程序。我们应该测试什么？

需要运行ipconfig命令，并检查是否执行失败。

你认为这里应该填什么？

你可以在不同的机器上用不同命令查看网络配置。在Linux和Mac上，你可以用一个叫/sbin/ifconfig的程序；而在Windows上可以用ipconfig的命令，它的路径保存在命令路径中。

下面这个程序试图运行/sbin/ifconfig程序，如果失败就运行ipconfig命令。你不用传递任何参数给这两条命令，仔细考虑将使用什么类型的exec()命令？

```
#include <stdio.h>
#include <unistd.h>
#include <errno.h>
#include <string.h>

int main()
{
  if ( execl("/sbin/ifconfig", "/sbin/ifconfig", NULL) == -1 )
    if (execlp("ipconfig", "ipconfig", NULL) == -1 ) {
      fprintf(stderr, "Cannot run ipconfig: %s", strerror(errno));
      return 1;
    }
  return 0;
}
```

- #include <unistd.h>：为了使用exec()函数，你需要它。
- #include <errno.h>：为了使用errno变量，你需要它。
- #include <string.h>：有了它，就能用strerror()函数显示错误消息了。
- 使用execl()，因为你有程序文件的路径。
- 如果execl()返回-1，就表明它执行失败，我们应该去找ipconfig。
- 我们可以用execlp()根据PATH查找ipconfig命令。
- 检查返回值是否是-1，以防命令执行失败。
- strerror()函数将显示任何可能出现的错误。

这里没有蠢问题

问： **system()不是比exec()简单吗？**

答： 是的，但操作系统必须解释你传给system()的字符串，这可能引发错误，尤其当你动态创建命令字符串时。

问： **为什么有那么多的exec()函数？**

答： 人们想以不同的方式创建进程，于是创建了不同版本的exec()来提高灵活性。

问： **为什么一定要检查系统调用的返回值？这样程序岂不是会很长？**

答： 如果在进行系统调用时不检查错误，代码是短了，但可能引发更多的错误。最好在最初写代码时就考虑到错误，以后找起错来也简单。

问： **调用了exec()函数以后还能做其他事吗？**

答： 不能，只要让exec()函数执行成功，就会修改进程。它会运行新程序替代你的程序。也就是说，只要exec()函数一运行，你的程序就会停止运行。

要点

- 系统调用是操作系统中的函数。
- 当进行系统调用时，相当于调用你程序外面的代码。
- system()系统调用可以运行命令字符串。
- system()用起来方便，但也容易出错。
- exec()系统调用在运行程序时给了你更多控制权。
- exec()系统调用有很多版本。
- 系统调用出错时通常会返回-1，但不是绝对的。
- 系统调用在出错的同时将errno变量设为错误码。

弄乱的消息

星巴克的员工写了一个新的订单生成程序，他们管它叫**coffee**：

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *w = getenv("EXTRA");
  if (!w)
    w = getenv("FOOD");
  if (!w)
    w = argv[argc - 1];
  char *c = getenv("EXTRA");
  if (!c)
    c = argv[argc - 1];
  printf("%s with %s\n", c, w);
  return 0;
}
```

为了检验程序，他们创建了这个测试程序。你能把代码片段和它们对应的输出结果连接起来吗？

```
#include <string.h>
#include <stdio.h>
#include <errno.h>
int main(int argc, char *argv[]){
```

候选代码从这里开始。

```
  fprintf(stderr,"Can't create order: %s\n", strerror(errno));
    return 1;
  }
  return 0;
}
```

候选代码：

将候选代码与对应的输出连接起来。

对应的输出：

```
char *my_env[] = {"FOOD=coffee", NULL};
if(execle("./coffee", "./coffee", "donuts", NULL, my_env) == -1){
  fprintf(stderr,"Can't run process 0: %s\n", strerror(errno));
  return 1;
}
```

```
char *my_env[] = {"FOOD=donuts", NULL};
if(execle("./coffee", "./coffee", "cream", NULL, my_env) == -1){
  fprintf(stderr,"Can't run process 0: %s\n", strerror(errno));
  return 1;
}
```

```
if(execl("./coffee", "./coffee", NULL) == -1){
  fprintf(stderr,"Can't run process 0: %s\n", strerror(errno));
  return 1;
}
```

```
char *my_env[] = {"FOOD=donuts", NULL};
if(execle("./coffee", "coffee", NULL, my_env) == -1){
  fprintf(stderr,"Can't run process 0: %s\n", strerror(errno));
  return 1;
}
```

```
coffee with donuts
```

```
cream with donuts
```

```
donuts with coffee
```

```
coffee with coffee
```

弄乱的消息解答

星巴克的员工写了一个新的订单生成程序，他们管它叫**coffee**：

```
#include <stdio.h>
#include <stdlib.h>

int main(int argc, char *argv[])
{
  char *w = getenv("EXTRA");
  if (!w)
    w = getenv("FOOD");
  if (!w)
    w = argv[argc - 1];
  char *c = getenv("EXTRA");
  if (!c)
    c = argv[argc - 1];
  printf("%s with %s\n", c, w);
  return 0;
}
```

为了检验程序，他们创建了这个测试程序。你能把代码片段和它们对应的输出结果连接起来吗？

候选代码从这里开始。

```
#include <string.h>
#include <stdio.h>
#include <errno.h>
int main(int argc, char *argv[]){



  fprintf(stderr,"Can't create order: %s\n", strerror(errno));
    return 1;
  }
  return 0;
}
```

候选代码：

```
char *my_env[] = {"FOOD=coffee", NULL};
if(execle("./coffee", "./coffee", "donuts", NULL, my_env) == -1){
  fprintf(stderr,"Can't run process 0: %s\n", strerror(errno));
  return 1;
}
```

```
char *my_env[] = {"FOOD=donuts", NULL};
if(execle("./coffee", "./coffee", "cream", NULL, my_env) == -1){
  fprintf(stderr,"Can't run process 0: %s\n", strerror(errno));
  return 1;
}
```

```
if(execl("./coffee", "coffee", NULL) == -1){
  fprintf(stderr,"Can't run process 0: %s\n", strerror(errno));
  return 1;
}
```

```
char *my_env[] = {"FOOD=donuts", NULL};
if(execle("./coffee", "./coffee", NULL, my_env) == -1){
  fprintf(stderr,"Can't run process 0: %s\n", strerror(errno));
  return 1;
}
```

对应的输出：

```
coffee with donuts
```

```
cream with donuts
```

```
donuts with coffee
```

```
coffee with coffee
```

用RSS读新闻

RSS源是网站发布新闻的常用方式。RSS源其实就是一个XML文件，里面有新闻的摘要和链接。当然，你完全有能力写一个直接从网页读取RSS文件的C程序，但这涉及一些你没有接触过的编程概念。为什么不找一个程序帮忙处理RSS文件呢？

动手吧！

RSS Gossip脚本下载地址：
https://github.com/dogriffiths/rssgossip/zipball/master。

如果你没有安装过Python，可以从这里下载：
http://www.python.org/

我想要死亡睡衣乐队的所有新闻。

编辑

RSS Gossip是一个Python小脚本，它可以根据某个关键字在RSS源中查找新闻。你必须先安装Python才能运行这个脚本，一旦有了Python和*rssgossip.py*，就可以像这样搜索新闻：

在unix环境中运行。

需要创建一个放RSS源地址的环境变量。

用搜索关键字运行rssgossip脚本。

这个源是虚构的，应该到网上找一个换掉它。

File Edit Window Help ReadAllAboutIt

```
> export RSS_FEED=http://www.cnn.com/rss/celebs.xml
> python rssgossip.py 'pajama death'
Pajama Death launch own range of kitchen appliances.
Lead singer of Pajama Death has new love interest.
"I never ate the bat" says Pajama Death's Hancock.
```

哈！我想到一个好主意，干嘛不写一个可以同时搜索多个RSS源的程序呢？你能写出来吗？

练习

编辑希望程序一次搜索多个RSS源，为此你可以为不同的RSS源多次运行*rssgossip.py*。幸运的是，兼职演员已经为你开了个头，但他们不会用exec()执行*rssgossip.py*脚本。为了运行脚本需要做哪些事？好好想想，然后完成**newshound**程序的代码。

为了节约纸张，这里省略了#include代码。

```
int main(int argc, char *argv[])
{
  char *feeds[] = {"http://www.cnn.com/rss/celebs.xml",
                   "http://www.rollingstone.com/rock.xml",
                   "http://eonline.com/gossip.xml"};
  int times = 3;
  char *phrase = argv[1];
  int i;
  for (i = 0; i < times; i++) {
    char var[255];
    sprintf(var, "RSS_FEED=%s", feeds[i]);
    char *vars[] = {var, NULL};
    if (..................("/usr/bin/python", "/usr/bin/python",
        ............................................................) == -1) {
      fprintf(stderr, "Can't run script: %s\n", strerror(errno));
      return 1;
    }
  }
  return 0;
}
```

这些是编辑钦点的RSS源（你可能想要用自己的）。

我们把搜索关键字当做参数传递。

遍历RSS源。

环境变量数组。

在编辑的Mac上，Python安装在这个位置。

需要在这里插入函数名。

需要在这里插入函数的其他参数。

newshound.c

想拿附加分？请回答……

程序运行时会做什么？

练习解答

编辑希望程序一次搜索多个RSS源，为此你可以为不同的RSS源多次运行*rssgossip.py*。幸运的是，**兼职演员**已经为你开了个头，但他们不会用exec()执行*rssgossip.py*脚本。为了运行脚本需要做哪些事？好好想想，然后完成**newshound**程序的代码。

```
int main(int argc, char *argv[])
{
  char *feeds[] = {"http://www.cnn.com/rss/celebs.xml",
                   "http://www.rollingstone.com/rock.xml",
                   "http://eonline.com/gossip.xml"};
  int times = 3;
  char *phrase = argv[1];
  int i;
  for (i = 0; i < times; i++) {
    char var[255];
    sprintf(var, "RSS_FEED=%s", feeds[i]);
    char *vars[] = {var, NULL};
    if ( execle ("/usr/bin/python", "/usr/bin/python",
         "./rssgossip.py", phrase, NULL, vars ) == -1) {
      fprintf(stderr, "Can't run script: %s\n", strerror(errno));
      return 1;
    }
  }
  return 0;
}
```

你要用参数列表和环境变量，所以是execLE。

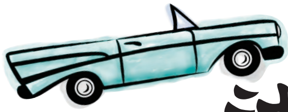
这是Python脚本的名字。

这是搜索关键字，以命令行参数传递。

以参数传递环境变量。

newshound.c

当运行程序时它会做什么呢？

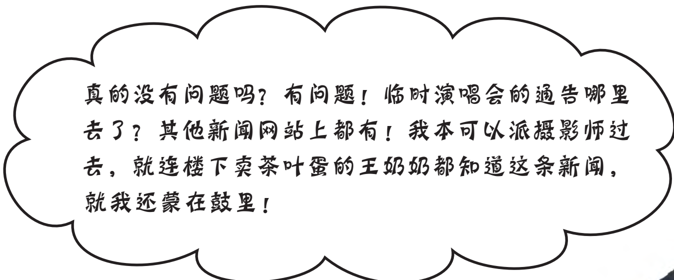


试驾

当你编译并运行程序时，看起来没什么问题：

```
File  Edit  Window  Help  ReadAllAboutIt
> ./newshound 'pajama death'
Pajama Death ex-drummer tells all.
New Pajama Death album due next month.
```

newshound程序让*rssgossip.py*脚本使用了RSS源数组中的数据。



程序其实有问题。

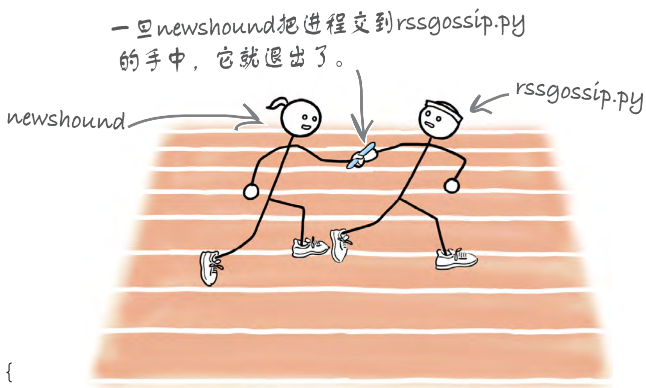
newshound程序虽然运行了*rssgossip.py*脚本，但它并没有为所有RSS源都运行脚本。它实际上只显示了列表中第一条RSS源的新闻，而与搜索关键字匹配的其他新闻都不见了踪影。

脑力风暴

再看一遍newshound程序，想一下它是怎么工作的。为什么它没能为第一条RSS源以外的其他RSS源运行*rssgossip.py*脚本？

exec()是程序中最后一行代码

exec()函数通过运行新程序来替换当前程序，那原来的程序去哪儿了？它终止了，而且是立刻终止，这就是为什么程序只为第一条RSS源运行了*rssgossip.py*脚本。程序在第一次调用execle()以后newshound程序就终止了。



循环只会运行一次。

```
for (i = 0; i < times; i++) {
  ...
  if (execle("/usr/bin/python", "/usr/bin/python",
             "./rssgossip.py", phrase, NULL, vars) == -1) {
  ...
  }
}
```

一旦调用execle()，整个程序就退出了。

与Unix和Mac不同，Windows天生不支持fork()。

如果想在Windows中使用fork()，必须先要安装Cygwin。

如果你想在启动另一个进程的同时让原进程继续运行下去，该怎么做？

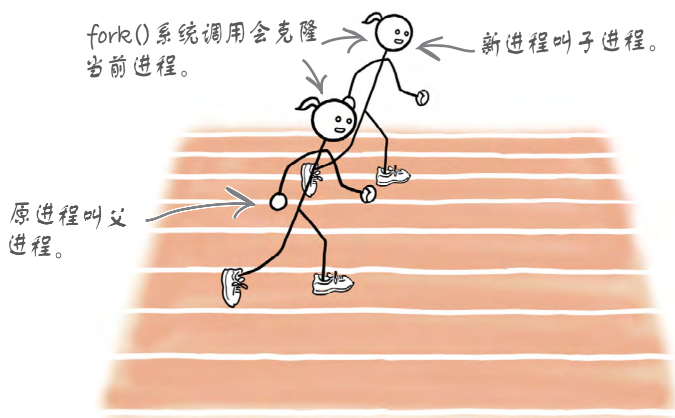
用fork()克隆进程

你可以用一个叫**fork()**的系统调用来解决这个问题。

fork()会克隆当前进程。新建副本将从同一行开始运行相同程序，变量和变量中的值完全一样，只有进程标识符（PID）和原进程不同。

原进程叫**父进程**，而新建副本叫**子进程**。

克隆当前进程如何能解决exec()的问题？我们来看看。



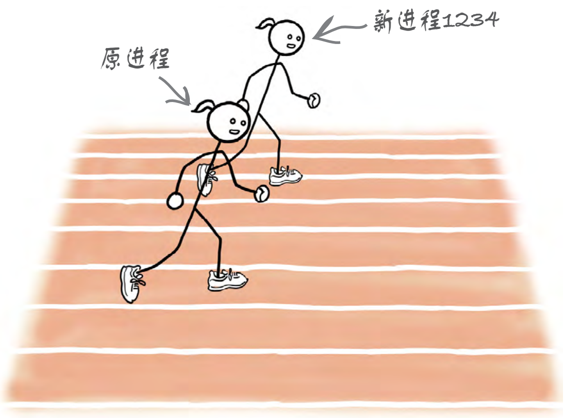
用fork()+exec()运行子进程

诀窍是在子进程中调用exec()函数，这样原来的父进程就能继续运行了。我们一步一步来看。

1. 复制进程

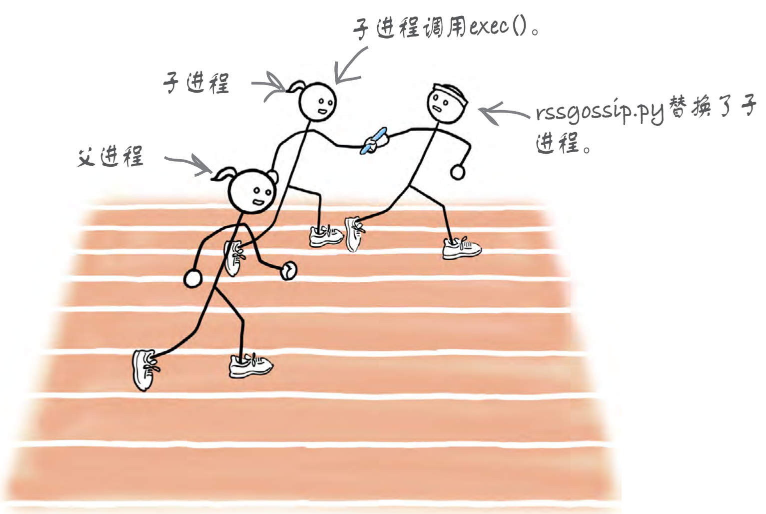
第一步用fork()系统调用复制当前进程。

进程需要以某种方式区分自己是父进程还是子进程，为此fork()函数向子进程返回0，向父进程返回**非零值**。



2. 如果是子进程，就调用exec()

这一刻，你有两个完全相同的进程在运行，它们使用相同的代码，但子进程（从fork()接收到0的那个）现在需要调用exec()运行程序替换自己：



现在你有两个独立的进程：子进程在运行*rssgossip.py*脚本，而原来的父进程可以继续做其他事，完全不受干扰。

代码冰箱贴

下面就来修改newshound程序。代码需要在独立进程中为每条RSS源运行*rssgossip.py*脚本。我们缩减了代码，你只需关注主循环即可。记得检查错误，千万别把父进程和子进程搞混了！

把冰箱贴贴在这个地方。

```
for (i = 0; i < times; i++) {
  char var[255];
  sprintf(var, "RSS_FEED=%s", feeds[i]);
  char *vars[] = {var, NULL};











}
```

叉子函数

你可以像这样调用fork()：

```
pid_t pid = fork();
```

fork()会返回一个整型值：为子进程返回0，为父进程返回一个正数。父进程将接收到子进程的进程标识符。

什么是**pid_t**？不同操作系统用不同的整数类型保存进程ID，有的用short，有的用int，操作系统使用哪种类型，pid_t就设为哪个。

```
fprintf(stderr, "Can't fork process: %s\n", strerror(errno));
```

```
fprintf(stderr, "Can't run script: %s\n", strerror(errno));
```

```
if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",
        phrase, NULL, vars) == -1) {
```

```
return 1;
```

```
pid_t pid = fork();
```

```
}
```

```
if (pid == -1) {
```

```
if (!pid) {
```

```
}
```

```
}
```

```
return 1;
```

代码冰箱贴解答

下面就来修改newshound程序。代码需要在独立进程中为每条RSS源运行*rssgossip.py*脚本。我们缩减了代码，你只需关注主循环即可。记得检查错误，千万别把父进程和子进程搞混了！

```
for (i = 0; i < times; i++) {
  char var[255];
  sprintf(var, "RSS_FEED=%s", feeds[i]);
  char *vars[] = {var, NULL};
  pid_t pid = fork();
```

首先，调用fork()克隆进程。

如果fork()返回-1，就说明在克隆进程时出了问题。

```
  if (pid == -1) {
    fprintf(stderr, "Can't fork process: %s\n", strerror(errno));
    return 1;
  }
```

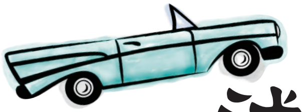
如果fork()返回0，说明代码运行在子进程中。

相当于if(pid==0)。

```
  if (!pid) {
```

如果你执行到这里，说明你是子进程，应该调用exec()运行脚本。

```
    if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",
               phrase, NULL, vars) == -1) {
      fprintf(stderr, "Can't run script: %s\n", strerror(errno));
      return 1;
    }
  }
}
```

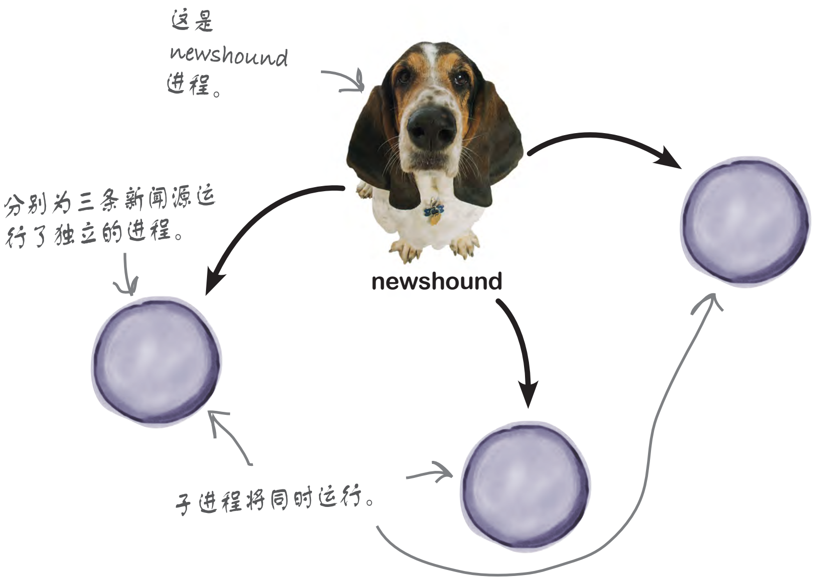


试驾

现在编译并运行代码，将看到：

```
File Edit Window Help ReadAllAboutIt
> ./newshound 'pajama death'
Pajama Death ex-drummer tells all.
New Pajama Death album due next month.
Photos from the surprise Pajama Death concert.
Official Pajama Death pajamas go on sale.
"When Pajama Death jumped the shark" by HenryW.
Breaking News: Pajama Death attend premiere.
```

通过克隆自己，然后在独立进程中运行Python脚本，newshound成功地为每个RSS源运行了独立的进程，而且最妙的是这些进程将**同时**运行。



这要比逐条读取新闻源快多了。通过学习用fork()和exec()创建并运行独立进程，不但能更好地利用现有软件，而且还能提高程序的性能。

这里没有蠢问题

问： **system()能在独立进程中运行程序吗？**

答： 可以，但使用system()使你没办法控制程序的运行方式。

问： **克隆进程岂不是很慢？我的意思是在用exec()替换子进程前我们还要等fork()复制完整个进程。**

答： 为了让fork进程变快，操作系统使用了很多技巧。比如操作系统不会真的复制父进程的数据，而是让父子进程共享数据。

问： **这样一来，如果子进程修改了存储器中的数据，岂不是会把事情搞砸？**

答： 不会，如果操作系统发现子进程要修改存储器，就会为它复制一份。

问： **这技术听起来真酷，有名字吗？**

答： 有，叫“写时复制”（copy-on-write）。

问： **pid_t就是int吗？**

答： 这取决于平台，你唯一知道的就是它是整型。

问： **我在int里保存fork()调用的结果，程序还是能运行。**

答： 最好还是用pid_t来保存进程ID，否则当把代码拿到其他机器上编译时可能会出错。

问： **为什么Windows不支持fork()系统调用？**

答： Windows管理进程的方式和其他操作系统完全不同，那些用来提高fork()效率的方法在Windows上很难实现，这可能就是为什么Windows没有内置的fork()。

问： **但Cygwin能让我在Windows中调用fork()，对吗？**

答： 是的，为了让Windows的进程看起来和Linux、Unix和Mac的一样，写Cygwin的专家做了很多工作。但由于他们还是需要依靠Windows来创建底层进程，所以Cygwin上的fork()要比其他平台上的fork()慢一些。

问： **如果我想让代码能在Windows上运行，有其他替代品吗？**

答： 嗯，有一个名叫CreateProcess()的函数，它是一个加强版的system()。如果你想了解更多信息，可以到http://msdn.microsoft.com搜索CreateProcess。

问： **多个新闻源的输出为什么不会混在一起？**

答： 操作系统会确保每个字符串都完整地打印出来。

要点

- 系统调用是内核中的函数。
- exec()函数比system()提供了更多控制权。
- exec()函数替换当前进程。
- fork()函数复制当前进程。
- 系统调用在失败时通常返回－1。
- 系统调用失败以后会把errno变量设为错误码。

C语言工具箱

你已经学完了第9章，现在你的工具箱又加入了进程和系统调用。关于本书的提示工具条的完整列表，请见附录ii。

system()会把字符串当成命令运行。

execl() = 参数列表
execle() = 参数列表 + 环境变量
execlp() = 参数列表 + 搜索PATH
execv() = 参数数组
execve() = 参数数组 + 环境变量
execvp() = 参数数组 + 搜索PATH

fork()复制当前进程。

fork()+exec()创建子进程。

10 进程间通信

沟通的艺术

创建进程只是个开始。

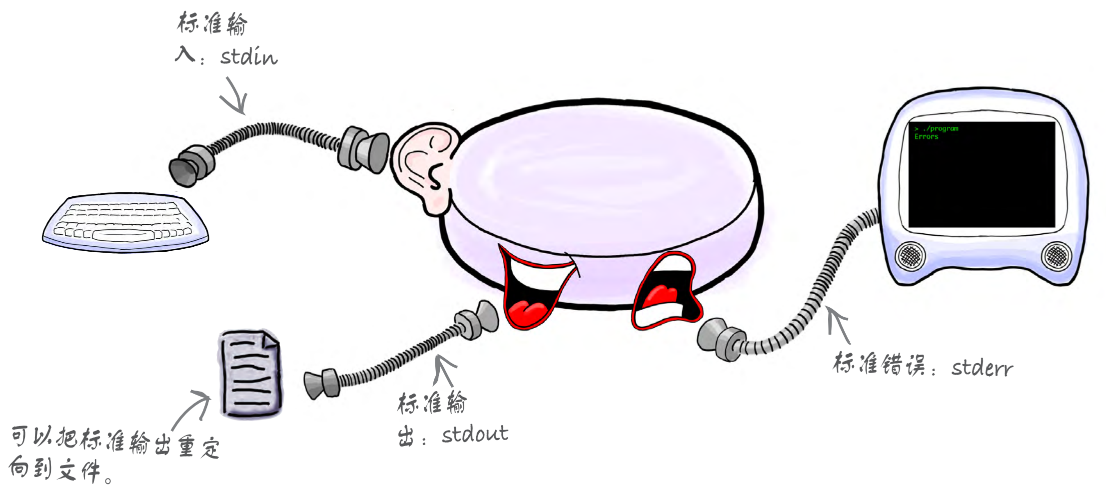
如果你想控制运行中的进程，向进程发送数据或读取它的输出，该怎么办？通过进程间通信，进程可以合力完成某件工作。我们将向你展示如何让程序与系统中其他程序通信，从而提升它的战斗力。

输入输出重定向

在命令行运行程序时，可以用“>”运算符把标准输出重定向到文件：

```
python ./rssgossip.py Snooki > stories.txt
```

可以用“>”运算符重定向输出。



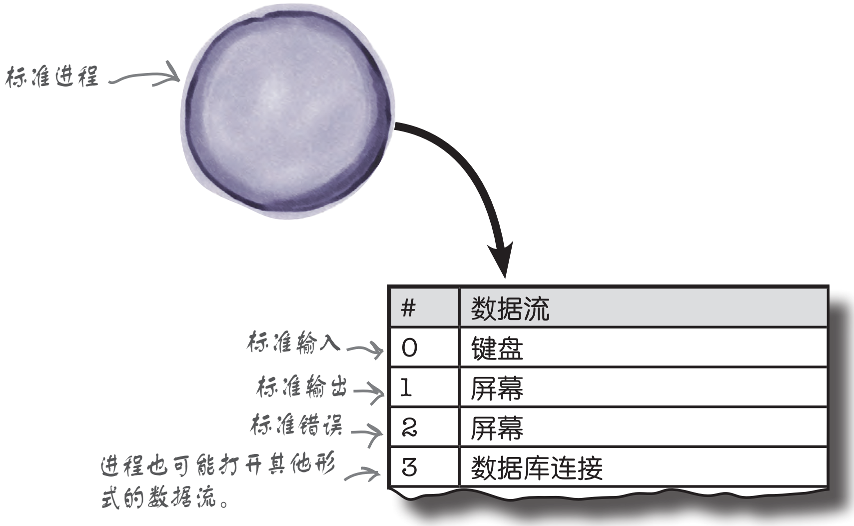
标准输出是三大默认**数据流**之一。顾名思义，数据流就是流动的数据，数据从一个进程流出，然后流入另一个进程。除了标准输入、标准输出和标准错误，还有其他形式的数据流，例如文件连接和网络连接也属于数据流。重定向进程的输出，相当于改变进程发送数据的方向。原来标准输出会把数据发送到屏幕，现在可以让它把数据发送到文件。

在命令行中，重定向是非常有用的命令。但有没有办法让进程重定向自己呢？

进程内部一瞥

进程含有它正在运行的程序，还有栈和堆数据空间。除此之外，进程还需要记录数据流的连向，比如标准输出连到了哪里。进程用文件描述符表示数据流，所谓的描述符其实就是一个数字。进程会把文件描述符和对应的数据流保存在描述符表中。

文件描述符是一个数字，它代表一条数据流。



#	数据流
0	键盘
1	屏幕
2	屏幕
3	数据库连接

描述符表的一列是文件描述符号，另一列是它们对应的数据流。虽然名字叫文件描述符，但它们不一定连接硬盘上的某个文件，也有可能连接键盘、屏幕、文件指针或网络。

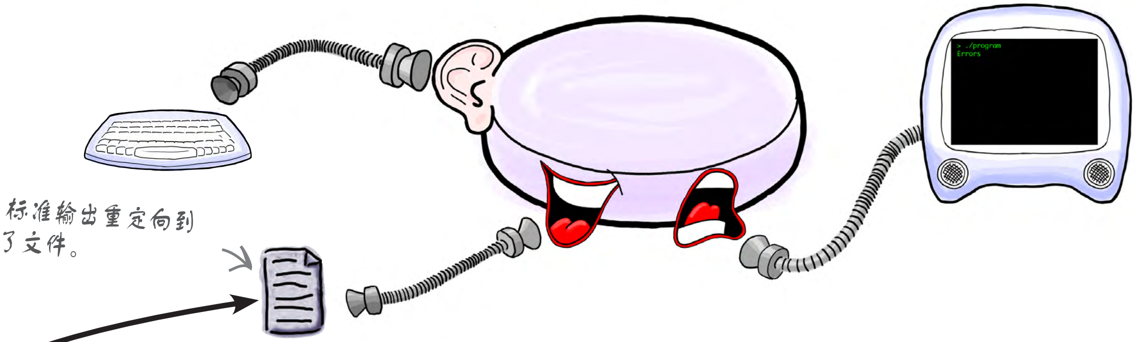
描述符表的前三项万年不变：0号标准输入，1号标准输出，2号标准错误。其他项要么为空，要么连接进程打开的数据流。比如程序在打开文件进行读写时，就会占用其中一项。

文件描述符描述的不一定是文件。

创建进程以后，标准输入连到键盘，标准输出和标准错误连到屏幕。它们会保持这样的连接，直到有人把它们重定向到了其他地方。

重定向即替换数据流

标准输入/输出/错误在描述符表中的位置是固定的，但它们指向的数据流可以改变。



也就是说，如果想重定向标准输出，只需要修改表中1号描述符对应的数据流就行了。

#	数据流
0	键盘
1	~~屏幕~~（stories.txt文件）
2	屏幕
3	数据库连接

所有向标准输出发送数据的函数会先查看描述符表，看1号描述符指向哪条数据流，然后再把数据写到这条数据流中，printf()便是如此。

进程可以重定向自己

到目前为止，你只在命令行中用“>”和“<”运算符重定向过程序，但只要修改描述符表，进程也能重定向它们自己。

百宝箱

难怪要用“2>”

你可以在命令行用“>”运算符重定向标准输出，用“2>”重定向标准错误：

```
./myprog > output.txt 2> errors.log
```

现在，知道为什么标准错误要用“2>”来重定向了吧，因为2是标准错误在描述符表中的编号。在很多操作系统中，也可以用“1>”来重定向标准输出。而在类Unix操作系统中，可以用以下命令把标准错误和标准输出重定向到一个地方：

```
./myprog 2>&1
```

“2>”表示“重定向标准错误”。

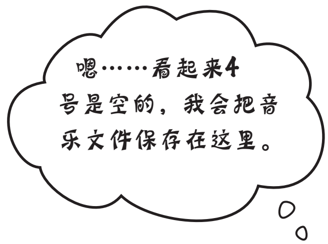
“&1”表示“到标准输出”。

fileno()返回描述符号

每打开一个文件，操作系统都会在描述符表中新注册一项。假设你打开了某个文件：

```
FILE *my_file = fopen("guitar.mp3", "r");
```

操作系统会打开*guitar.mp3*文件，然后返回一个指向它的指针，操作系统还会遍历描述符表寻找空项，把新文件注册在其中。



#	数据流
0	键盘
1	屏幕
2	屏幕
3	数据库连接
4	guitar.mp3文件

那么如何根据文件指针知道它是几号描述符呢？答案是调用**fileno()**函数。

```
int descriptor = fileno(my_file);
```

它会返回4。

在失败时不返回－1的函数很少，fileno()就是其中之一。只要你把打开文件的指针传给fileno()，它就一定会返回描述符编号。

dup2()复制数据流

每次打开文件都会使用描述符表中新的一项。但如果你想修改某个已经注册过的数据流，比如想让3号描述符重新指向其他数据流，该怎么做？可以用**dup2()**函数，dup2()可以复制数据流。假设你在4号描述符中注册了*guitar.mp3*文件指针，下面这行代码就能同时把它连接到3号描述符：

```
dup2(4, 3);
```

#	数据流
0	键盘
1	屏幕
2	屏幕
3	数据库连接(guitar.mp3文件)
4	guitar.mp3文件

虽然*guitar.mp3*文件只有一个，与它相连的数据流也只有一条，但数据流（即FILE*）同时注册在了文件描述符3和4中。

既然你已经学会了如何在描述符表中查找文件和修改数据流，也就能把进程的标准输出重定向到某个文件。

还在为错误代码烦恼？

每次你在系统调用时都需要反反复复写那些错误处理代码。还犹豫什么！赶快使用我们的独家秘方，我们将向你展示如何重用错误代码，从此你将告别重复代码：

下面两段代码一看头就大：

```
pid_t pid = fork();
if (pid == -1) {
  fprintf(stderr, "无法克隆进程: %s\n", strerror(errno));
  return 1;
}

if (execle(...) == -1) {
  fprintf(stderr, "无法运行脚本: %s\n", strerror(errno));
  return 1;
}
```

重复的代码会带来不必要的编码压力。

有没有办法可以消除重复代码呢？当然有！只要创建一个error()函数，就可以一劳永逸。

error()函数是什么？这些return怎么处理？总不见得也移到error()函数里吧？

不需要！exit()系统调用是结束程序的最快方式。完全不用操心怎么返回主函数，直接调用exit()，你的程序就会灰飞烟灭！

首先， 需要把处理代码放到一个单独的error()函数中，然后把return语句换成exit()系统调用。

为了使用exit系统调用，必须包含stdlib.h头文件。

```
void error(char *msg)
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(1);
}
```

exit(1)会立刻终止程序，并把退出状态置1。

现在就可以把那些烦人的错误检查代码换成：

```
pid_t pid = fork();
if (pid == -1) {
  error("无法克隆进程");
}

if (execle(...) == -1) {
  error("无法运行脚本");
}
```

这么做简单多了！

警告：每次程序执行只有一次调用exit()的机会，“程序突然结束恐惧症”患者慎用。

磨笔上阵

程序把*rssgossip.py*脚本的输出保存到*stories.txt*文件中。程序只搜索一个RSS源，其他都和newshound一样。程序少了一行把子进程的标准输出重定向到*stories.txt*的代码，你能补出来吗？你可能用到描述符表的知识。

为了节省空间，我们省去了#include和error()函数。

```
int main(int argc, char *argv[])
{
  char *phrase = argv[1];
  char *vars[] = {"RSS_FEED=http://www.cnn.com/rss/celebs.xml", NULL};
  FILE *f = fopen("stories.txt", "w");
  if (!f) {
    error("Can't open stories.txt");
  }
  pid_t pid = fork();
  if (pid == -1) {
    error("Can't fork process");
  }
  if (!pid) {
    if (.........................................................................) {
      error("Can't redirect Standard Output");
    }
    if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",
                         phrase, NULL, vars) == -1) {
      error("Can't run script");
    }
  }
  return 0;
}
```

如果不能以“写”模式打开stories.txt，f就是0。

我们将用事先写好的error()函数报告错误。

这里应该填什么？

newshound2.c

磨笔上阵 解答

程序把*rssgossip.py*脚本的输出保存到*stories.txt*文件中。程序只搜索一个RSS源，其他都和newshound一样。程序少了一行把子进程的标准输出重定向到*stories.txt*的代码，凭借对描述符表的了解，你把它补了出来。

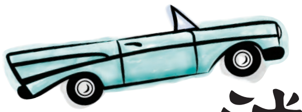
```
int main(int argc, char *argv[])
{
  char *phrase = argv[1];
  char *vars[] = {"RSS_FEED=http://www.cnn.com/rss/celebs.xml", NULL};
  FILE *f = fopen("stories.txt", "w");   <- 以“写”模式打开stories.txt。
  if (!f) {   <- 如果f是0，说明无法打开文件。
    error("Can't open stories.txt");
  }
  pid_t pid = fork();
  if (pid == -1) {
    error("Can't fork process");
  }
                  这段代码会修改子进程，因为pid是0.
  if (!pid) {                                  令1号描述符指向stories.txt文件。
    if (dup2(fileno(f), 1) == -1) {
      error("Can't redirect Standard Output");
    }
    if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",
                         phrase, NULL, vars) == -1) {
      error("Can't run script");
    }
  }
  return 0;
}
```

newshound2.c

你写对了吗？程序把子进程（脚本程序）的描述符表改成了这样：

也就是说当*rssgossip.py*把数据发往标准输出时，数据应该出现在*stories.txt*文件中。

#	数据流
0	键盘
1	stories.txt文件
2	屏幕
3	stories.txt文件



试驾

编译运行程序，将看到：

运行程序。

显示stories.txt文件的内容。

在Windows上需要运行Cygwin。

```
File Edit Window Help ReadAllAboutIt
> ./newshound2 'pajama death'
> cat stories.txt
Pajama Death ex-drummer tells all.
New Pajama Death album due next month.
```

新闻保存在stories.txt文件中。

发生了什么事?

当程序用`fopen()`打开*stories.txt*文件时，操作系统把文件`f`注册到了描述符表中，`fileno(f)`是文件`f`使用的描述符编号，而`dup2()`函数设置了标准输出描述符（1号），让它也指向了该文件。

文件中没有数据? TMD，咋回事?

透明的

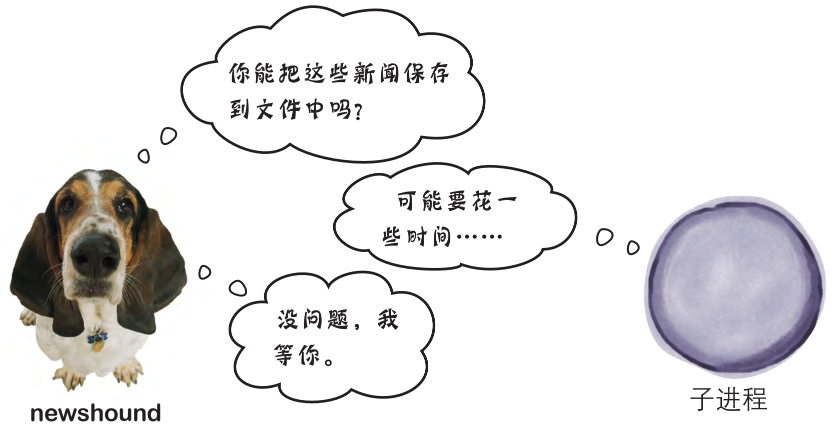
```
File Edit Window Help ReadAllAboutIt
> ./newshound2 'pajama death'
> cat stories.txt
>
```

脑力风暴

假设RSS源中的确有你要找的新闻，可为什么程序结束以后*stories.txt*还是空的?

有时需要等待……

newshound2程序启用独立的进程运行*rssgossip.py*脚本，而子进程一创建就和父进程没关系了。*rssgossip.py*还没有完成任务，newshound2程序就结束了，所以*stories.txt*还是空的。也就是说，操作系统必须提供一种方式，让你等待子进程完成任务。



waitpid()函数

waitpid()函数会等子进程结束以后才返回，也就是说可以在程序中加几行代码，让它等到*rssgossip.py*脚本运行结束以后才退出。

需要包含sys/wait.h头文件。

```
#include <sys/wait.h>
```

这个变量用来保存进程信息。

int指针。

新代码加到newshound2程序底部。

可以在这里加选项。

```
  int pid_status;
  if (waitpid(pid, &pid_status, 0) == -1) {
    error("等待子进程时发生了错误");
  }
  return 0;
}
```

进程号

newshound2.c

waitpid()聚焦

waitpid()接收三个参数：

waitpid(pid, pid_status, options)

- **pid**
  父进程在克隆子进程时会得到子进程的ID。

- **pid_status**
  pid_status用来保存进程的退出信息。因为waitpid()需要修改pid_status，因此它必须是个指针。

- **选项**
  waitpid()有一些选项，详情可以输入man waitpid查看。如果把选项设为0，函数将等待进程结束。

什么是pid_status?

waitpid()函数结束等待时会在pid_status中保存一个值，它告诉你进程的完成情况。为了得到子进程的退出状态，可以把pid_status的值传给**WEXITSTATUS()**宏：

```
if (WEXITSTATUS(pid_status))    如果退出状态不是0
  puts("Error status non-zero");
```

为什么要用宏来查看？因为pid_status中保存了好几条信息，只有前8位表示进程的退出状态，可以用宏来查看这8位的值。

试驾

运行newshound2程序，它会在退出前检查*rssgossip.py*脚本是否完成：

newshound2运行后，stories.txt中出现了新闻。

```
File  Edit  Window  Help  ReadAllAboutIt
> ./newshound2 'pajama death'
> cat stories.txt
Pajama Death ex-drummer tells all.
New Pajama Death album due next month.
```

在程序中加入waitpid()很容易办到，它可以让代码更加可靠。而在此之前无法确定子进程是否已经写完了文件，也就是说newshound2并不是一个合格的工具，你无法在脚本中使用它，也无法为它创建界面。

重定向输入、输出，然后让进程相互等待，**进程间通信**就这么简单。一旦进程可以合作——通过共享数据和互相等待——它们将所向披靡。

要点

- exit()可以快速结束程序。
- 所有打开的文件都记录在描述符表中。
- 通过修改描述符表就可以重定向输入和输出。
- fileno()能在表中查找描述符。
- dup2()可以用来修改描述符表。
- waitpid()等待进程结束。

这里没有蠢问题

问： 用**exit()**来结束程序比从**main()**返回更快吗？

答： 不会。但如果你已经调用了exit()，就不需要想办法让代码再回到main()函数。在你调用exit()的一瞬间，程序就升天了。

问： 为了防止调用失败，调用**exit()**时需要检查它的返回值是否为－1吗？

答： 不需要，exit()不会失败，因此也就没有返回值。exit()是唯一没有返回值而且不会失败的函数。

问： 传给**exit()**的那个数字是退出状态吗？

答： 是的。

问： 标准输入、标准输出和标准错误一定是描述符表的0、1、2号吗？

答： 一定。

问： 每当我打开一个新文件，它都会自动添加到描述符表中吗？

答： 没错。

问： 通常是几号描述符？

答： 新文件总是按序加入描述符表，如果第一个空的描述符是4号，你的文件就会用它。

问： 描述符表有多大？

答： 从0号到255号。

问： 描述符表那么麻烦，有必要用吗？

答： 当然有，不使用描述符表，就不能改变程序的工作方式，也就不能重定向。

问： 除了用标准输出，还有没有其他方法把数据发送到屏幕？

答： 在一些系统上，比如Unix，如果打开*/dev/tty*文件，就可以把数据直接发送到终端。

问： 我能用**waitpid()**等待其他进程吗？还是只有我启动的那些？

答： 你可以用waitpid()等待任何进程。

问： 为什么不能根据**wait_pid(..., &pid_status, ...)**中的**pid_status**直接判断退出状态？

答： 因为pid_status中还包含了其他信息。

问： 哪些信息？

答： 如果一个进程自然死亡，WIFSIGNALED(pid_status)就为假，如果是他杀，WIFSIGNALED(pid_status)就为真。

问： **pid_status**明明是整型变量，怎么可以包含多条信息？

答： 用不同的位来保存不同的信息。pid_status的前8位保存了退出状态，而其他信息保存在了剩余那些位中。

问： 是不是只要自行提取出**pid_status**的前8位，就可以不用**WEXITSTATUS()**？

答： 最好还是用WEXITSTATUS()，它不但可以提高代码的可读性，而且无论int在你的平台上有多大，程序都能正确工作。

问： 为什么**WEXITSTATUS()**要大写？

答： 因为WEXITSTATUS()是宏，不是函数。编译器运行时会把宏替换为一小段代码。

家书抵万金

你已经学会了用exec()和fork()运行独立进程，也知道怎么把子进程的输出重定向到文件，但如果你想从子进程直接获取数据呢？在进程运行时**实时**读取它生成的数据，而不是等子进程把所有数据都发送到文件，再从文件中把数据读取出来，有这个可能吗？

从rssgossip读取新闻链接

为了举这个例子，*rssgossip.py*脚本提供了一个-u选项，可以用它显示找到新闻的URL：

-u吩咐脚本在显示新闻时附上链接。

```
File  Edit  Window  Help
> python rssgossip.py -u 'pajama death'
Pajama Death ex-drummer tells all.
    http://www.rock-news.com/exclusive/24.html
New Pajama Death album due next month.
    http://www.rolling-stone.com/pdalbum.html
```

URL以tab开头。

新闻URL。

你完全可以运行脚本，然后把它的输出保存在文件中，不过这样很慢。如果父子进程能够在子进程运行期间通信，速度会快很多。

用管道连接进程

你曾用过某样东西实时连接两个进程，那就是管道。

两个进程通过管道连接。

grep过滤脚本的输出。

rssgossip.py把输出发送到管道。

```
File Edit Window Help ReadAllAboutIt
python rssgossip.py -u 'pajama death' | grep 'http'
    http://www.rock-news.com/exclusive/24.html
    http://www.rolling-stone.com/pdalbum.html
```

可以在命令行用管道把一个进程的**输出**连接到另一个进程的**输入**。在这个例子中，你手动运行了*rssgossip.py*脚本，然后把它的输出传给了**grep**命令，grep找出了包含**http**的那些行。

管道两侧的命令是父子关系

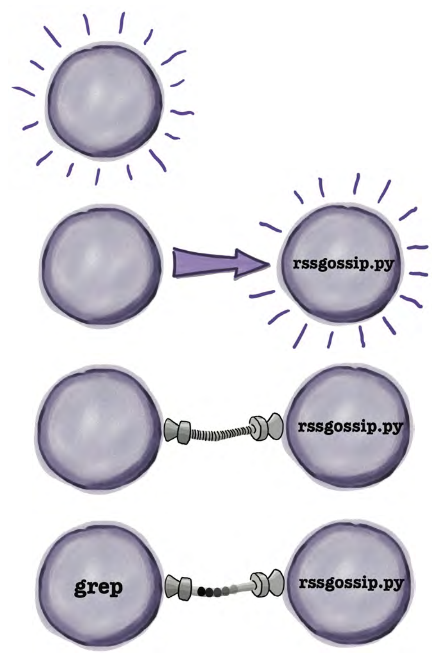
当你在命令行用管道连接两条命令时，实际把它们当成了父子进程来连接，在上面的例子中，grep命令是*rssgossip.py*脚本的父进程。

1. 命令行创建了父进程。

2. 父进程在子进程中克隆出了rssgossip.py脚本。

3. 父进程用管道把子进程的输出连接到自己的输入。

4. 父进程运行了grep命令。



管道常用来在命令行中连接两个进程。但如果想要在C代码中连接两个进程呢？怎么才能给子进程连接管道，实时读取它的输出？

案例研究：在浏览器中打开新闻

假设你想在浏览器中打开*rssgossip.py*脚本找到的新闻链接，你将在父进程中运行程序，在子进程中运行*rssgossip.py*。需要创建管道，把*rssgossip.py*的输出和程序的输入连接起来。

如何创建管道?

pipe()打开两条数据流

因为子进程需要把数据发送到父进程，所以要用管道连接子进程的标准输出和父进程的标准输入。你将用**pipe()**函数建立管道。还记得吗？我们说过，每当打开数据流时，它都会加入描述符表。pipe()函数也是如此，它创建两条相连的数据流，并把它们加到表中，然后只要你往其中一条数据流中写数据，就可以从另一条数据流中读取。

#	数据流
0	标准输入
1	标准输出
2	标准错误
3	管道读取端
4	管道写入端

fd[0] → 3

fd[1] → 4

调用pipe()创建这两个描述符。

pipe()在描述符中创建这两项时，会把它们的文件描述符保存在一个包含两个元素的数组中：

描述符将保存在这个数组中。

把数组名传递给pipe()函数。

```
int fd[2];
if (pipe(fd) == -1) {
  error("Can't create the pipe");
}
```

pipe()函数创建了管道，并返回了两个描述符：fd[1]用来向管道**写**数据，fd[0]用来从管道**读**数据，你将在父、子进程中使用这两个描述符。

fd[1]写管道；fd[0]读管道。

子进程

在子进程中，需要关闭管道的fd[0]端，然后修改子进程的标准输出，让它指向描述符fd[1]对应的数据流。

子进程不会从管道中读取数据。

```
close(fd[0]);
dup2(fd[1], 1);
```

关闭管道的读取端。

子进程把管道写入端连到标准输出。

fd[0]是管道的读取端。

#	数据流
0	标准输入
1	~~标准输出~~（管道写入端）
2	标准错误
3	~~管道读取端~~
4	管道写入端

子进程不会从管道读取数据……

但它会写数据。

fd[1]是管道的写入端。

也就是说，子进程发送给标准输出的数据都会写到管道中。

父进程

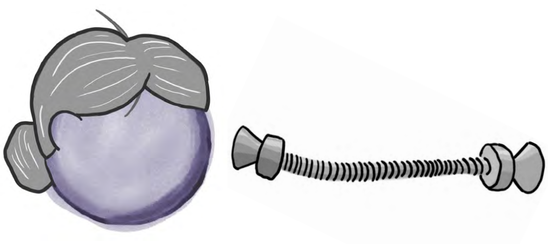
在父进程中，需要关闭管道的fd[1]端（不需要写），然后重定向父进程的标准输入，让它从描述符fd[0]对应的数据流中读取数据：

父进程把读取端连到标准输入。

```
dup2(fd[0], 0);
close(fd[1]);
```

fd[0]是管道的读取端。

关闭管道的写入端。



#	数据流
0	~~标准输入~~（管道读取端）
1	标准输出
2	标准错误
3	管道读取端
4	~~管道写入端~~

父进程将从管道读取数据……

……但不会写。

子进程写到管道的数据将由父进程的标准输入读取。

在浏览器中打开网页

程序需要用机器上的浏览器打开网页，但不同操作系统与程序交互的方式不同，因此实现起来多少有些难度。

好在兼职演员已经为你写好了这部分的代码，它能够在绝大多数系统上打开网页。但他们好像还有要事在身，所以只采用了最简单的实现方式：

代码熟食

```
void open_url(char *url)
{
  char launch[255];
  sprintf(launch, "cmd /c start %s", url);
  system(launch);
  sprintf(launch, "x-www-browser '%s' &", url);
  system(launch);
  sprintf(launch, "open '%s'", url);
  system(launch);
}
```

在Windows上打开网页。

在Linux上打开网页。

在Mac上打开网页。

代码用了三条命令，它们分别可以在Windows、Linux和Mac上打开URL。每次都有两条命令会失败，但只要有一条成功就行了。

滑野雪

你能写出比兼职演员更好的代码吗？为何不用fork()和exec()在你喜欢的操作系统中重写这部分代码呢？

练习

大部分代码已经写好了，你只需要填写用管道连接父子进程的那部分。为了节约空间，我们去掉了#include语句、error()和open_url()函数。别忘了，在这个程序中是子进程对父进程说话，所以请以正确的方式连接管道！

```
int main(int argc, char *argv[])
{
  char *phrase = argv[1];
  char *vars[] = {"RSS_FEED=http://www.cnn.com/rss/celebs.xml", NULL};
  int fd[2];
  ..........................................................
  ..........................................................
  ..........................................................
  pid_t pid = fork();
  if (pid == -1) {
    error("Can't fork process");
  }
  if (!pid) {
    ..........................................................
    ..........................................................
    if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",
                    "-u", phrase, NULL, vars) == -1) {
      error("Can't run script");
    }
  }
  ..........................................................
  ..........................................................
  char line[255];
  while (fgets(line, 255, ..............................)) {
    if (line[0] == '\t')
      open_url(line + 1);
  }
  return 0;
}
```

可以换成其他RSS源。（指向 http://www.cnn.com/rss/celebs.xml）

这个数组将保存管道的描述符。（指向 int fd[2];）

在这里创建管道。

是父进程还是子进程？这里应该写什么？

-u让脚本显示新闻URL。

这里你在父进程还是子进程中呢？需要对管道做什么？

这里填什么？要从哪里读取数据？

如果line以tab开头……

……就说明它是URL。

line+1是tab字符以后的字符串。

news_opener.c

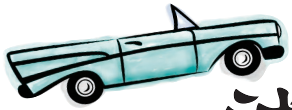
练习解答

大部分代码已经写好了，你只需要填写用管道连接父子进程的那部分。为了节约空间，我们去掉了#include语句、error()和open_url()函数。

```
int main(int argc, char *argv[])
{
  char *phrase = argv[1];
  char *vars[] = {"RSS_FEED=http://www.cnn.com/rss/celebs.xml", NULL};
  int fd[2];        创建管道，并把描述符保存在fd[0]和fd[1]中。
    if (pipe(fd) == -1) {
      error("Can't create the pipe");     为了防止管道创建失败，需要检查pipe()的返回值。
    }
  pid_t pid = fork();
  if (pid == -1) {
    error("Can't fork process");
  }                 这里你在子进程中。
  if (!pid) {
    dup2(fd[1], 1);     把标准输出设为管道的写入端。
    close(fd[0]);       子进程不会读取管道，所以我们将关闭读取端。
    if (execle("/usr/bin/python", "/usr/bin/python", "./rssgossip.py",
                          "-u", phrase, NULL, vars) == -1) {
      error("Can't run script");
    }
  }                 从这里开始你就在父进程中。
  dup2(fd[0], 0);       把标准输入重定向到管道的读取端。
  close(fd[1]);         关闭管道的写入端，因为父进程不需要向管道写数据。
  char line[255];
  while (fgets(line, 255, stdin)) {
    if (line[0] == '\t')
      open_url(line + 1);
  }
  return 0;
}
```

stdin：将从标准输入读取数据，因为管道连到了标准输入。 ← 也可以用fd[0]。

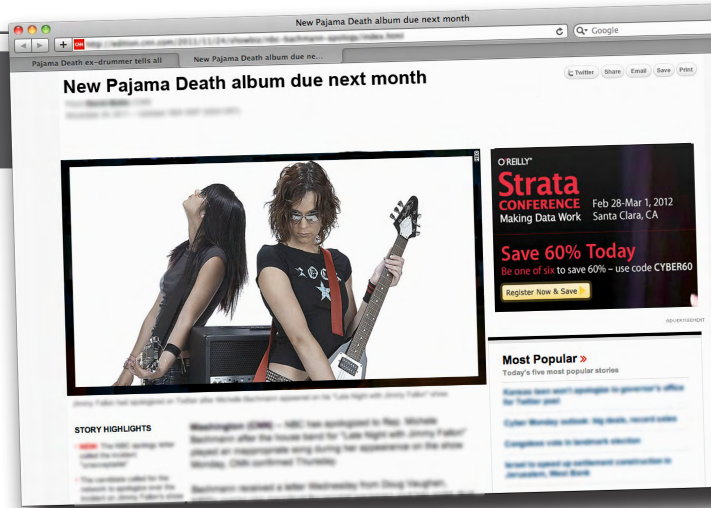
news_opener.c



试驾

编译代码并运行程序，出现了：

```
File Edit Window Help ReadAllAboutIt
> ./news_opener 'pajama death'
```



程序在浏览器中打开了所有找到的新闻。

太棒了，程序工作了。

news_opener程序在一个独立的进程中运行了*rssgossip.py*，并让它显示找到新闻的URL。所有本来应该发送到屏幕上的输出现在通过管道重定向到news_opener父进程，news_opener就可以在浏览器中打开新闻了。

管道是连接进程的好办法。现在你不但能够运行进程，控制它们的环境，而且还能获取进程的输出，这样就可以实现很多功能。任何一个能够在命令行中运行的程序你都可以在C代码中调用并控制它。

滑野雪

既然你已经知道了如何控制*rssgossip.py*，为什么不试着控制一些其他程序呢？在类Unix机器或任何使用Cygwin的Windows机器中可以获取以下程序：

curl/wget

可以用这两个程序与网络服务器通信，也可以在C代码中使用它们与网络通信。

mail/mutt

可以在命令行用这两个程序发送邮件。如果在机器上装了它们，C程序就能发送邮件。

convert

convert命令可以转换图片格式。你可以写一个C程序，它输出文本格式的SVG图表，然后用convert命令把SVG转化成PNG图片。

这里没有蠢问题

问： 管道是文件吗？

答： 这取决于操作系统创建管道的方式，通常用`pipe()`创建的管道都不是文件。

问： 就是说也有可能是文件？

答： 你可以创建基于文件的管道，它们通常叫有名管道或FIFO（First In First Out，先进先出）文件。

问： 它能干嘛？

答： 因为基于文件的管道有名字，所以两个进程只要知道管道的名字也能用它来通信，即使它们非父子进程。

问： 太好了！怎么使用有名管道？

答： 使用`mkfifo()`系统调用，详情请见http://tinyurl.com/cdf6ve5。

问： 如果不用文件来实现管道，那用什么？

答： 通常用存储器。数据写到存储器中的某个位置，然后再从另一个位置读取。

问： 如果我试图读取一个空的管道会怎么样？

答： 程序会等管道中出现东西。

问： 父进程如何知道子进程什么时候结束？

答： 子进程结束时，管道会关闭。`fgets()`将收到EOF（End Of File，文件结束符），于是`fgets()`函数返回0，循环就结束了。

问： 父进程能对子进程说话吗？

答： 当然可以。你完全可以反向连接管道，让数据从父进程发送到子进程。

问： 管道能够双向通信吗？这样父子进程不就可以边听边讲了？

答： 管道只能单向通信。不过可以创建两个管道，一个从父进程连到子进程，另一个从子进程连到父进程。

要点

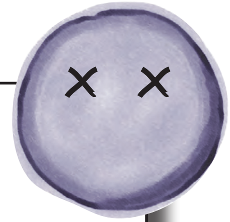
- 父子进程可以用管道通信。
- `pipe()`函数创建一个管道和两个描述符。
- 一个描述符是管道的读取端，另一个是写入端。
- 可以把标准输入和标准输出重定向到管道。
- 父子进程各自使用管道的一端。

进程之死

创建进程、配置环境和进程间通信，这些你都见过了，但你知道进程是怎么“死”的吗？比如你的程序正在从键盘读取数据，这时用户按了Ctrl-C，程序就停止运行了。

为什么会这样？从输出来看，程序还没执行到第二个printf()就已经退出了，所以Ctrl-C叫停的不仅仅是fgets()，而是整个程序。是操作系统停止了程序？还是fgets()函数调用了exit()？这中间到底发生了什么？

```
#include <stdio.h>

int main()
{
  char name[30];
  printf("Enter your name: ");
  fgets(name, 30, stdin);
  printf("Hello %s\n", name);
  return 0;
}
```

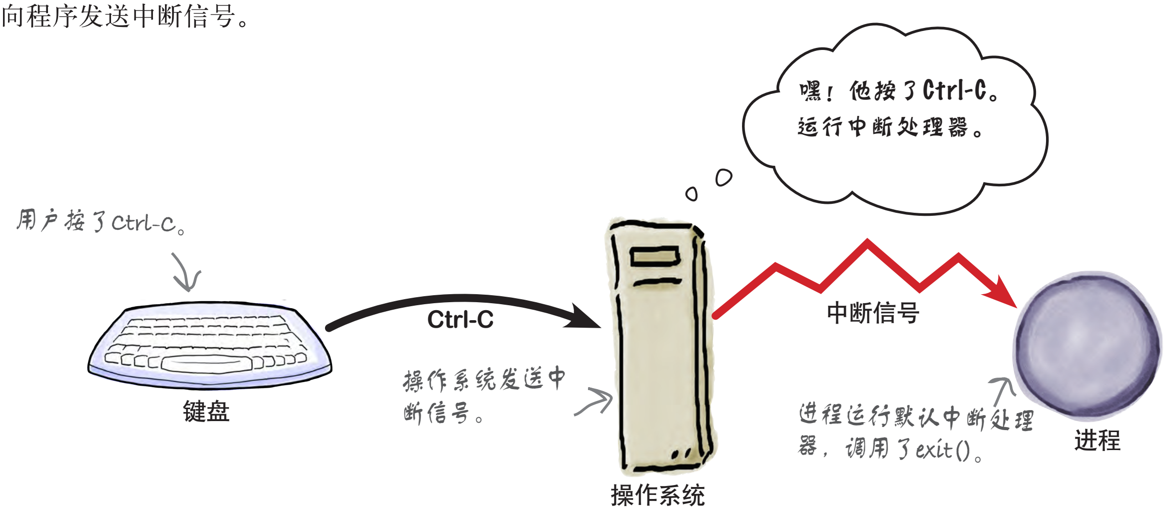


```
File Edit Window Help
> ./greetings
Enter your name: ^C
>
```

只要按Ctrl-C，程序就会停止运行，为什么会这样？

操作系统用信号控制程序

奥秘发生在操作系统中。当调用fgets()函数时，操作系统会从键盘读取数据，但当它看到用户按了Ctrl-C，就会向程序发送中断信号。



信号是一条短消息，即一个整型值。当信号到来时，进程必须停止手中一切工作去处理信号。进程会查看信号映射表，表中每个信号都对应一个信号处理器函数。中断信号的默认信号处理器会调用exit()函数。

操作系统为什么不直接结束程序，而是要在信号表中查找信号？因为这样就可以在进程接收到信号时运行你自己的代码。

信号映射表。

信号	处理函数
SIGURG	不做事情
SIGINT	调用exit()

中断信号。

SIGINT的值是2.

默认信号处理器会调用exit()。

捕捉信号然后运行自己的代码

有时你希望在别人打断你的程序时运行自己的代码。假设进程打开了一些文件连接或网络连接，你希望在退出之前把它们关闭，并且做一些清理工作。当计算机在向你发送信号时，如何让它运行你的代码呢？可以用**sigaction**。

sigaction是一个函数包装器

sigaction是一个结构体，它有一个函数指针。sigaction告诉操作系统进程收到某个信号时应该调用哪个函数。如果想在某人向进程发送中断信号时让操作系统调用diediedie()函数，就需要把diediedie()函数包装成sigaction。

sigaction的创建方法如下：

```
struct sigaction action;
action.sa_handler = diediedie;
sigemptyset(&action.sa_mask);
action.sa_flags = 0;
```

创建新动作。

想让计算机调用哪个函数。

sigaction包装起来的那个函数就叫处理器。

一些附加标志位，将它们置0就行了。

用掩码来过滤sigaction要处理的信号。

通常会用一个空的掩码。

sigaction包装的函数就叫处理器，因为它将用来处理发送给它的信号，而处理器必须以特定的方式创建。

处理器必须接收信号参数

信号是一个整型值，如果你创建了一个自定义处理器函数，就需要接收一个整型参数，像这样：

```
void diediedie(int sig)
{
  puts ("Goodbye cruel world....\n");
  exit(1);
}
```

这是处理程序捕捉到的信号编号。

因为我们以参数的形式传递信号，所以多个信号可以共用一个处理器；也可以为每个信号写一个处理器，完全由你做主。

处理器的代码应该短而快，刚好能处理接收到的信号就好。

在处理器函数中使用标准输出和标准错误时要小心。

虽然示例代码在标准输出中显示了文本，但在更复杂的程序中这么做时千万要小心。之所以会有信号就是因为程序中发生了故障，而故障可能就是标准输出无法使用，因此要小心。

用sigaction()来注册sigaction

创建sigaction以后，需要用**sigaction()**函数来让操作系统知道它的存在：

```
sigaction(signal_no, &new_action, &old_action);
```

sigaction()接收如下三个参数。

- **信号编号。**
这个整型值代表了你希望处理的信号。通常会传递SIGINT或SIGQUIT这样的标准信号。 ← 过一会儿你将了解更多关于标准信号的信息。

- **新动作。**
你想注册的新sigaction的**地址**。

- **旧动作。**
如果你想保存被替换的信号处理器，可以再传一个sigaction指针；如果不想保存，可以设置为NULL。

如果sigaction()函数失败，会返回－1，并设置errno变量。为了缩短代码的长度，书中的一些代码会跳过错误检查，但你一定要在自己的代码中检查错误。

代码熟食

下面这个函数简化了将函数注册为信号处理器的过程：

```
int catch_signal(int sig, void (*handler)(int))
{
  struct sigaction action;
  action.sa_handler = handler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  return sigaction (sig, &action, NULL);
}
```

- 信号编号（sig）
- 处理器指针（(*handler)(int)）
- 创建动作。（struct sigaction action;）
- 将动作处理器设为我们传进来的函数。（action.sa_handler = handler;）
- 使用一个空的掩码。（sigemptyset(&action.sa_mask);）
- 返回sigaction()的值，这样就能检查错误了。（return sigaction (sig, &action, NULL);）

只要把信号编号和函数名传给catch_signal()，就能设置信号处理器了。

```
catch_signal(SIGINT, diediedie)
```

使用信号处理器

现在程序已经可以在用户按下Ctrl-C以后做一些事情了。

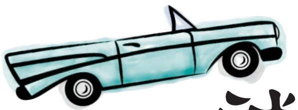
```
#include <stdio.h>
#include <signal.h>     ← 需要包含signal.h头文件。
#include <stdlib.h>
                 ↙ 新的信号处理器。
void diediedie(int sig)    ← 操作系统把信号传给处理器。
{
  puts ("Goodbye cruel world....\n");
  exit(1);
}
               ↙ 注册处理器的函数。
int catch_signal(int sig, void (*handler)(int))
{
  struct sigaction action;
  action.sa_handler = handler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  return sigaction (sig, &action, NULL);
}

int main()
{
  if (catch_signal(SIGINT, diediedie) == -1) {
    fprintf(stderr, "Can't map the handler");
    exit(2);
  }
  char name[30];
  printf("Enter your name: ");
  fgets(name, 30, stdin);
  printf("Hello %s\n", name);
  return 0;
}
```

处理器返回类型为void。→ (void diediedie)

SIGINT表示我们要捕捉中断信号。

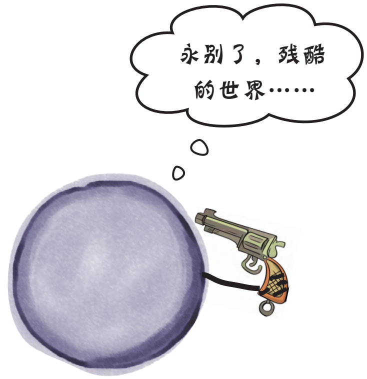
将中断处理程序设为diediedie()函数。

程序要求用户输入名字，然后它会等待输入。如果用户没有输入名字而是按了Ctrl-C，操作系统会自动向进程发送中断信号（SIGINT），然后用我们在catch_signal()函数中注册的sigaction来处理这个信号。sigaction中有一个指向diediedie()函数的指针，程序会调用这个函数，显示消息并调用exit()。



试驾

运行新版程序，然后按Ctrl-C，结果如下：




```
File Edit Window Help
> ./greetings
Enter your name: ^CGoodbye cruel world....
>
```

操作系统收到了Ctrl-C以后向进程发送`SIGINT`信号，然后进程运行了你的`diediedie()`函数。

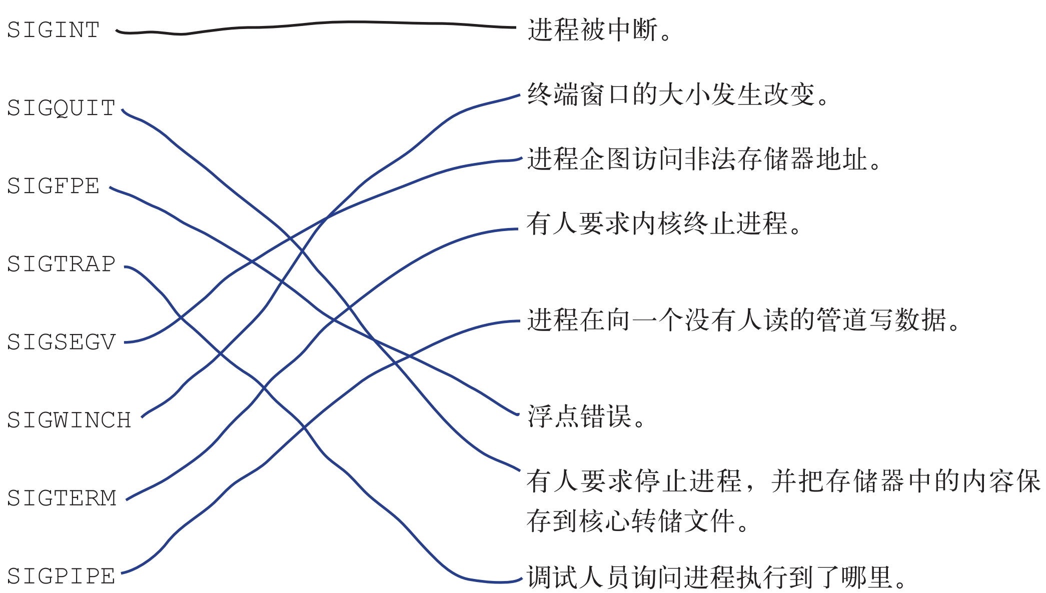
猜猜我是谁？

操作系统可以向进程发送各种信号，请把下列信号与引起它们的原因连接起来。

SIGINT	进程被中断。
SIGQUIT	终端窗口的大小发生改变。
SIGFPE	进程企图访问非法存储器地址。
	有人要求内核终止进程。
SIGTRAP	
SIGSEGV	进程在向一个没有人读的管道写数据。
SIGWINCH	浮点错误。
SIGTERM	有人要求停止进程，并把存储器中的内容保存到核心转储文件（core dump file）。
SIGPIPE	调试人员询问进程执行到了哪里。

猜猜我是谁？解答

操作系统可以向进程发送各种信号，你把下列信号与引起它们的原因连接了起来。



这里没有蠢问题

问： 如果中断信号的处理器不调用`exit()`，程序还能结束吗？

答： 不会。

问： 也就是说，我可以写一个忽略中断信号的程序？

答： 可以是可以，但这可不是什么好主意。程序收到错误信号以后最好还是退出，即使之前你运行了自己的代码。

用kill发送信号

怎么测试你写的信号处理代码呢？在类Unix操作系统中有一个叫**kill**的命令，之所以叫kill是因为这个命令通常用来“杀死”进程。事实上，kill只是向进程发送了一个信号，kill默认会向进程发送SIGTERM信号，你也可以用它发送其他信号。

在Windows上用Cygwin

我们打开两个终端试试。在一个终端运行程序，在另一个终端用kill向程序发送信号：

```
File  Edit  Window  Help
> ps -a
77868 ttys003    0:00.02 bash
78222 ttys003    0:00.01 ./testprog
> kill 78222
> kill -INT 78222
> kill -SEGV 78222
> kill -KILL 78222
```

ps显示当前进程。

向程序发送SIGTERM信号。

向程序发送SIGINT信号。

向程序发送SIGSEGV信号。

发送SIGKILL，进程不能忽略这个信号。

我们想把信号发到这个程序，78222是进程ID。

以上kill命令将向进程发送信号，然后运行进程中配置好的处理函数。但有一个例外，代码捕捉不到**SIGKILL**信号，也没法忽略它。也就是说，即使程序中有一个错误导致进程对任何信号都视而不见，还是能用kill -KILL结束进程。

SIGSTOP信号也是无法忽略的，它可以用来暂停进程。

kill -KILL <进程号> 一定可以送你的程序上西天。

用raise()发送信号

有时你想让进程向自己发送信号，这时就可以用raise()函数。

```
raise(SIGTERM);
```

通常会在自定义的信号处理函数中使用raise()，这样程序就能在接收到低级别的信号时引发更高级别的信号。

这叫**信号升级**。

打电话叫程序起床

当计算机中发生了进程需要知道的事情时，操作系统就会向进程发送信号。比如用户想中断进程或“杀死”进程，或进程企图做一件它不应该做的事情，比如访问受限存储器。

除了在发生错误时使用，有时进程也需要产生自己的信号，比如闹钟信号**SIGALRM**。闹钟信号通常由进程的间隔定时器创建。间隔定时器就像一台闹钟：你可以定一个时间，其间程序就会去做其他事情：

把闹钟调到120秒以后闹铃。

其间代码就会做其他事：

```
alarm(120);
do_important_busy_work();
do_more_busy_work();
```

尽管程序正忙着做其他事，计时器还是会在后台运行，120秒以后……

…定时器发出SIGALRM信号

当进程收到信号以后就会停止手中一切工作来处理信号。进程在收到闹钟信号以后默认会结束进程，但通常情况下使用定时器不是为了让它帮你“杀死”程序，而是为了利用闹钟信号的处理器去做另一件事：

用之前创建的catch_signal函数来捕捉信号。

```
catch_signal(SIGALRM, pour_coffee);
alarm(120);
```

闹钟信号可以实现多任务。如果需要每隔几秒运行一个任务，或者想限制花费在某个任务上的时间，就可以用闹钟信号让程序打断自己。

不要同时使用alarm()和sleep()。

sleep()函数会让程序沉睡一段时间。和alarm()函数一样，它也使用了间隔计时器，因此同时使用这两个函数会发生冲突。

重置信号与忽略信号

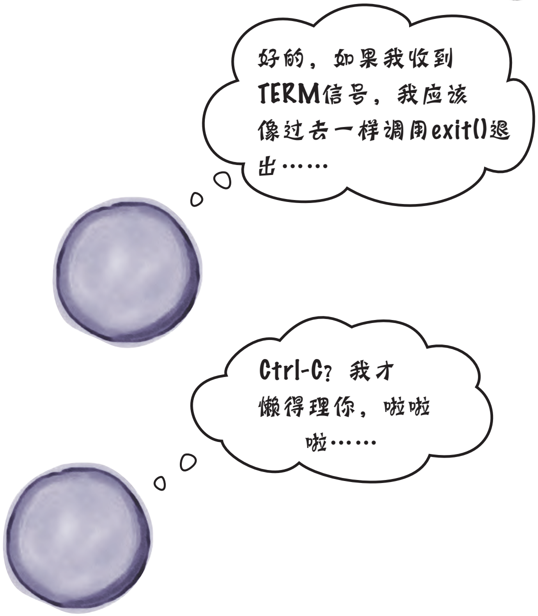
你已经见过如何设置自定义信号处理器了，但如果你想还原默认的信号处理器怎么办？*signal.h*头文件中有一个特殊的符号**SIG_DFL**，它代表以默认方式处理信号。

```
catch_signal(SIGTERM, SIG_DFL);
```

同时，你还可以用**SIG_IGN**符号让进程忽略某个信号。

```
catch_signal(SIGINT, SIG_IGN);
```

在你决定忽略某个信号前一定要慎重考虑，信号是控制进程和终止进程的重要方式，如果忽略了它们，程序就很难停下来。



这里没有蠢问题

问： 我能把闹钟定在几分之一秒后响铃吗？

答： 可以是可以，但很复杂。需要用另一个函数setitimer()，它可以把进程间隔计时器的单位设为几分之一秒。

问： 具体怎么做？

答： 详情请见http://tinyurl.com/3o7hzbm。

问： 为什么一个进程只有一个定时器？

答： 定时器由操作系统的内核管理，如果一个进程有很多定时器，内核就会变得很慢，因此操作系统需要限制进程能使用的定时器个数。

问： 定时器可以实现多任务？太好了，也就是说我能同时做几件事？

答： 非也。别忘了，进程在处理信号时会停止一切工作，也就是说一次只能做一件事，稍后会看到如何让代码同时做多件事。

问： 重复设置定时器会怎么样？

答： 每次调用alarm()函数都会重置定时器，也就是说如果把闹钟调到10秒，但过一会儿又把它设为了10分钟，那么闹钟信号10分钟以后才会触发，第一个10秒计时就失效了。

这个程序用来测试用户的数学水平，它要求用户做乘法。程序的结束条件如下：

1. 用户按了Ctrl-C。
2. 回答时间超过5秒。

程序在结束时会显示最终得分并把退出状态设为0。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

int score = 0;

void end_game(int sig)
{
  printf("\nFinal score: %i\n", score);
  ............................................................
}

int catch_signal(int sig, void (*handler)(int))
{
  struct sigaction action;
  action.sa_handler = handler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  return sigaction (sig, &action, NULL);
}
```

显示完得分应该干嘛？

```
void times_up(int sig)
{
  puts("\nTIME'S UP!");
  raise(........................................................);
}
```

引发什么信号？

```
void error(char *msg)
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(1);
}

int main()
{
  catch_signal(SIGALRM, ....................................);
  catch_signal(SIGINT, ......................................);
  srandom (time (0));
  while(1) {
    int a = random() % 11;
    int b = random() % 11;
    char txt[4];

    ....................................................................................
    printf("\nWhat is %i times %i? ", a, b);
    fgets(txt, 4, stdin);
    int answer = atoi(txt);
    if (answer == a * b)
      score++;
    else
      printf("\nWrong! Score: %i\n", score);
  }
  return 0;
}
```

catch_signal()函数会做什么？

确保每次都得到不同的随机数。

a、b是0到10的随机数。

嗯……这行写什么？看看说明书吧……

练习解答

这个程序用来测试用户的数学水平，它要求用户做乘法。程序的结束条件如下：

1. 用户按了Ctrl-C。
2. 回答时间超过5秒。

程序在结束时会显示最终得分并把退出状态设为0。

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
#include <time.h>
#include <string.h>
#include <errno.h>
#include <signal.h>

int score = 0;

void end_game(int sig)
{
  printf("\nFinal score: %i\n", score);
  exit(0);
}

int catch_signal(int sig, void (*handler)(int))
{
  struct sigaction action;
  action.sa_handler = handler;
  sigemptyset(&action.sa_mask);
  action.sa_flags = 0;
  return sigaction (sig, &action, NULL);
}
```

需要在结束程序的同时把退出状态设为0.

```
void times_up(int sig)
{
  puts("\nTIME'S UP!");
  raise(SIGINT);
}
```

引发SIGINT信号，让程序调用end_game()显示最后得分。

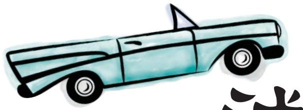
```
void error(char *msg)
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(1);
}

int main()
{
  catch_signal(SIGALRM, times_up);
  catch_signal(SIGINT, end_game);
  srandom (time (0));
  while(1) {
    int a = random() % 11;
    int b = random() % 11;
    char txt[4];
    alarm(5);
    printf("\nWhat is %i times %i? ", a, b);
    fgets(txt, 4, stdin);
    int answer = atoi(txt);
    if (answer == a * b)
      score++;
    else
      printf("\nWrong! Score: %i\n", score);
  }
  return 0;
}
```

signal()函数设置信号处理器。

确保每次都得到不同随机数。

将闹钟信号设为5秒后触发。

只要能在5秒内再次回到到这个地方，定时器就会重置，闹钟信号也不会触发。



试驾

为了测试这个程序，你需要多运行几遍。

测试一：按Ctrl-C

第一次先回答几个问题然后按Ctrl-C。

Ctrl-C向进程发送中断信号（SIGINT），程序显示最终得分然后调用exit()退出。

```
File Edit Window Help
> ./math_master
What is 0 times 1? 0
What is 6 times 1? 6
What is 4 times 10? 40
What is 2 times 3? 6
What is 7 times 4? 28
What is 4 times 10? ^C
Final score: 5
>
```

用户在这里按下了Ctrl-C。

程序在结束前显示了最终得分。

测试二：等5秒

第二次，不按Ctrl-C，而是在一个问题出现后等待至少5秒，看看会发生什么。

闹钟信号（SIGALRM）触发了。程序在等用户输入答案，但用户花了太长时间，定时器信号被发送给了程序。程序马上跳转到times_up()处理器，先显示了“TIME’S UP!”消息，然后把信号升级为SIGINT，于是程序显示出了最后的得分。

```
File Edit Window Help
> ./math_master
What is 5 times 9? 45
What is 2 times 8? 16
What is 9 times 1? 9
What is 9 times 3?
TIME'S UP!
Final score: 3
>
```

嗯……看起来他有些迟钝。

虽然信号有些复杂，但很好用。信号可以让程序从容结束，而间隔定时器可以帮助处理一些超时任务。

这里没有蠢问题

问： 信号按什么顺序发送，程序就会按什么顺序接收吗？

答： 如果两个信号发送间隔很短就不会，操作系统会先发送它认为更重要的信号。

问： 总是如此吗？

答： 取决于你的平台。例如在Cygwin的很多版本中，信号会按发送的顺序接收，但通常不应该做这样的假设。

问： 如果一个信号发送了两次，进程都会接收到吗？

答： 还是要看情况，在Linux和Mac中，如果一个信号在很短的时间里发送了两次，内核只会发送其中的一个；而在Cygwin中两个信号都会发送，但不应该做这样的假设。

要点

- 操作系统用信号来控制进程。
- 程序通常用信号来结束。
- 进程收到信号后会运行信号处理器。
- 大部分错误信号的默认处理器会终止程序。
- 可以用sigaction()函数替换处理器。
- 可以用raise()向自己发送信号。
- 间隔定时器发送SIGALRM信号。
- alarm()函数设置间隔定时器。
- 每个进程只能有一个定时器。
- 不要同时使用sleep()和alarm()。
- kill命令可以向进程发送信号。
- kill -KILL一定可以终止进程。

C语言工具箱

你已经学完了第10章，现在你的工具箱又加入了进程间通信。关于本书的提示工具条的完整列表，请见附录ii。

- exit()立即终止程序。
- fileno()查找描述符。
- dup2()复制数据流。
- waitpid()等待进程结束。
- pipe()创建通信管道。
- 进程可以用管道通信。
- 信号是O/S发出的消息。
- 用sigaction()处理信号。
- 程序可以用raise()向自己发送信号。
- kill命令发送信号。
- alarm()会在几秒钟以后向进程发送SIGALRM信号。

11 网络与套接字

金窝，银窝，不如127.0.0.1的草窝

不同计算机上的程序需要对话。

你已经学习了怎么用I/O与文件通信，还学习了如何让同一台计算机上的两个进程通信，现在你将走向世界舞台，让C程序通过互联网和世界各地的其他程序通信。本章的最后你将创建具有服务器和客户端功能的程序。

---

① BLAB是服务器连接网络四个步骤的首字母缩写（后文会提到），blab还有“唠叨”的含义。——译者注

互联网knock-knock服务器

互联网中大部分的底层网络代码都是用C语言写的。网络程序通常由两部分程序组成：服务器和客户端。

你将用C语言创建一个通过互联网说笑话的服务器。你可以这样在机器上启动服务器：

```
File Edit Window Help KnockKnock
> ./ikkp_server
Waiting for connection
```

除了告诉你它正在运行，服务器不会在屏幕上显示任何东西。可以再开一个控制台，使用一个叫telnet的客户端程序连接服务器。telnet接收两个参数：一个是服务器地址，另一个是服务器运行的端口。如果在运行服务器的那台计算机上运行telnet，地址可以填**127.0.0.1**：

为了测试服务器的代码，本章中你将多次使用telnet。

使用Windows自带的telnet可能会有问题，这是它与网络通信的方式所造成的。如果你安装的是Cygwin版的telnet，就没事。

30 000是网络端口号。

如果你在同一台计算机上运行服务器，就填127.0.0.1。

服务器已应答。

你这样回复道。

```
File Edit Window Help Who'sThere?
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Internet Knock-Knock Protocol Server
Version 1.0
Knock! Knock!
> Who's there?
Oscar
> Oscar who?
Oscar silly question, you get a silly answer
Connection closed by foreign host.
>
```

动手吧！

需要用telnet程序连接服务器。很多系统自带了telnet，可以用以下命令检查计算机上有没有telnet：

```
telnet
```

如果你的计算机上没有telnet，可以用以下方式安装：

Cygwin:
打开Cygwin的安装程序（`setup.exe`），搜索telnet。

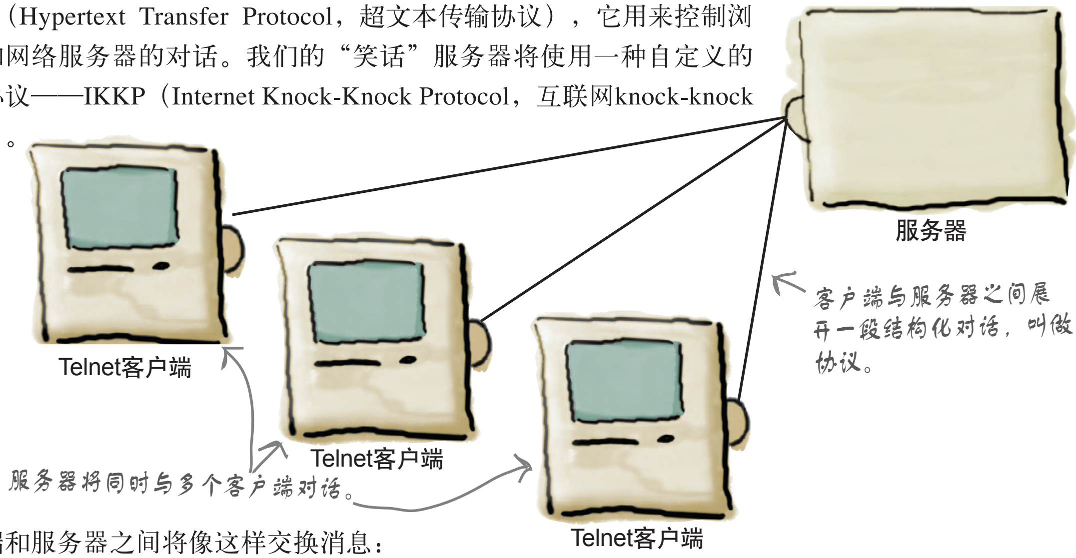
Linux:
在包管理器中搜索*telnet*，很多操作系统的包管理器叫新立得（synaptic）。

Mac:
如果没有telnet，可以从*www.macports.org*或*www.finkproject.org*安装。

knock-knock服务器概述

服务器将同时与多个客户端通信。客户端与服务器之间将展开一段结构化对话，叫做**协议**。互联网使用了各种协议，一部分是低层协议，另一部分是高层协议。低层协议有IP（Internet Protocol，网际协议），它用来控制二进制的0和1在互联网中的发送方式；高层协议有HTTP（Hypertext Transfer Protocol，超文本传输协议），它用来控制浏览器和网络服务器的对话。我们的“笑话”服务器将使用一种自定义的高层协议——IKKP（Internet Knock-Knock Protocol，互联网knock-knock协议）。

协议是一段结构化对话。



客户端和服务器之间将像这样交换消息：

协议通常有一套严格的规则。客户端和服务器都遵守这些规则就没事，但只要它们中有一方违反了规则，对话就会戛然而止。

② “敲门笑话”（knock-knock joke）是一种利用谐音制造笑点的笑话。讲笑话的人以“Knock knock！”开场，听到的人接“Who’s there?”，然后讲笑话的必须回答一个人名，比如“Oscar.”，对方继续问：“Oscar who?”，这时说笑话的人必须用Oscar开头造句，比如“Oscar silly question, you get a silly answer.”，这里的“Oscar”谐音“You ask”。——译者注

BLAB：服务器连接网络四部曲

为了与外界沟通，C程序用数据流读写字节。到目前为止，你用过三种数据流，它们分别连接的是文件、标准输入和标准输出。如果想要写一个与网络通信的程序，就需要一种新数据流——套接字。

```
#include <sys/socket.h>
...
int listener_d = socket(PF_INET, SOCK_STREAM, 0);
if (listener_d == -1)
  error("无法打开套接字");
```

需要包含这个头文件。

listener_d是套接字描述符。

互联网套接字。

这是协议号。填0就行了。

这是你在上一章创建的error()函数。

在使用套接字与客户端程序通信前，服务器需要历经四个阶段：绑定（Bind）、监听（Listen）、接受（Accept）和开始（Begin），首字母缩写为BLAB。

B代表绑定端口。
L代表监听。
A代表接受连接。
B代表开始通信。

1. 绑定端口

计算机可能同时运行多个服务器程序：一个发送网页，一个发送邮件，另一个运行聊天服务器。为了防止不同对话发生混淆，每项服务必须使用不同的**端口**（port）。端口就好比电视频道，我们在不同端口使用不同的网络服务，就像我们通过不同频道收看不同的电视节目。

服务器在启动时，需要告诉操作系统将要使用哪个端口，这个过程叫端口绑定。knock-knock服务器将使用30000端口，为了绑定它，你需要两样东西：套接字描述符和套接字名。套接字名是一个表示“互联网30000端口”的结构。

```
#include <arpa/inet.h>
...
struct sockaddr_in name;
name.sin_family = PF_INET;
name.sin_port = (in_port_t)htons(30000);
name.sin_addr.s_addr = htonl(INADDR_ANY);
int c = bind (listener_d, (struct sockaddr *) &name, sizeof(name));
if (c == -1)
  error("无法绑定端口");
```

为了创建互联网地址，需要包含这些头文件。

这些代码将创建一个表示“互联网30000端口”的套接字名。

2. 监听

如果你的笑话服务器出了名，可能会有很多客户端同时连接它。想让客户端排队等待连接吗？可以用listen()系统调用告诉操作系统你希望队列有多长。

```
if (listen(listener_d, 10) == -1)
  error("无法监听");
```

队列长度为10。

调用listen()把队列长度设为10，也就是说最多可以有10个客户端同时尝试连接服务器，它们不会立即得到响应，但可以排队等待，而第11个客户端会被告知服务器太忙。

前10个客户端排队等待。

第11个和第12个将被告知服务器太忙。

3. 接受连接

一旦绑定完端口，设置完监听队列，唯一可以做的就是等待。服务器一生都在等待有客户端来连接它们。accept()系统调用会一直等待，直到有客户端连接服务器时，它会返回第二个套接字描述符，然后就可以用它通信了。

```
struct sockaddr_storage client_addr;
unsigned int address_size = sizeof(client_addr);
int connect_d = accept(listener_d, (struct sockaddr *)&client_addr, &address_size);
if (connect_d == -1)
  error("无法打开副套接字");
```

client_addr将保存连接客户端的详细信息。

服务器将用新的**连接描述符**connect_d……

开始通信。

脑力练习

为什么accept()系统调用要创建一个新的套接字描述符？服务器为什么不用监听端口的那个套接字通信？

套接字不是传统意义上的数据流

到目前为止，你见过的数据流都一样，不管是连接文件的数据流，还是连接标准输入或输出的数据流，都可以用fprintf()和fscanf()与它们通信。但套接字有一点点不同，套接字是双向的，它既可以用作输入也可以用作输出，也就是说要用其他函数和它通信。

如果想向套接字输出数据，就要用**send()**函数，而不是fprintf()。

```
char *msg = "Internet Knock-Knock Protocol Server\r\nVersion 1.0\r\nKnock! Knock!\r\n> ";
if (send(connect_d, msg, strlen(msg), 0) == -1)
  error("send");
```

将通过网络发送这条消息。

套接字描述符。

消息和消息长度。

最后一个参数是高级选项，填0就行了。

记住：一定要检查系统调用的返回值，send()也不例外。网络错误随处可见，服务器必须处理它们。

百宝箱

如何选择端口号？

为服务器程序选择端口号时千万要小心。现如今有各式各样的服务器，所以不要选其他程序用过的端口号。在Cygwin和大多数Unix中有一个*/etc/services*文件，它列出了很多常用服务使用的端口号。在选择端口时必须确保没有其他程序用过。

端口号从0开始一直到65535，首先你需要决定用小号码（1024以下）还是大号码。很多计算机中，只有超级用户或管理员才有资格使用1024号以下的端口，因为小号的端口留给了一些知名服务，如网页服务器和邮件服务器。操作系统只允许管理员使用这些端口，防止普通用户启动一些多余的服务。

通常情况下，请使用1024号以上的端口。

磨笔上阵

下面这个服务器会为已连接的客户端随机提出忠告，但它少了很多系统调用。你需要把它们补全。再有，程序向客户端发送一条忠告，然后就结束了。其中有一段代码需要循环执行，请问是哪段？

为了节约空间，我们省略了include。

```
int main(int argc, char *argv[])
{
  char *advice[] = {
    "Take smaller bites\r\n",
     "Go for the tight jeans. No they do NOT make you look fat.\r\n",
     "One word: inappropriate\r\n",
     "Just for today, be honest. Tell your boss what you *really* think\r\n",
     "You might want to rethink that haircut\r\n"
  };
  int listener_d = ....................(PF_INET, SOCK_STREAM, 0);

  struct sockaddr_in name;
  name.sin_family = PF_INET;
  name.sin_port = (in_port_t)htons(30000);
  name.sin_addr.s_addr = htonl(INADDR_ANY);
  ....................(listener_d, (struct sockaddr *) &name, sizeof(name));

  ....................(listener_d, 10);
  puts("Waiting for connection");

  struct sockaddr_storage client_addr;
  unsigned int address_size = sizeof(client_addr);
  int connect_d = ...............(listener_d, (struct sockaddr *)&client_addr, &address_size);
  char *msg = advice[rand() % 5];

  ....................(connect_d, msg, strlen(msg), 0);
  close(connect_d);

  return 0;
}
```

如果想拿附加分，就补上#include语句，让程序能够正确运行。但程序员好像忘了一件事，到底是什么事呢？提示：注意系统调用。

程序员忘了 ....

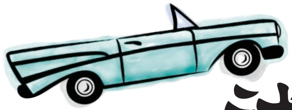
磨笔上阵 解答

下面这个服务器会为已连接的客户端随机提出忠告，但它少了很多系统调用。你需要把它们补全。再有，程序向客户端发送一条忠告，然后就结束了。其中有一段代码需要循环执行，请问是哪段？

```
int main(int argc, char *argv[])
{
  char *advice[] = {
    "Take smaller bites\r\n",
     "Go for the tight jeans. No they do NOT make you look fat.\r\n",
     "One word: inappropriate\r\n",
     "Just for today, be honest. Tell your boss what you *really* think\r\n",
     "You might want to rethink that haircut\r\n"
  };
  int listener_d = socket(PF_INET, SOCK_STREAM, 0);   ← 创建套接字。

  struct sockaddr_in name;
  name.sin_family = PF_INET;
  name.sin_port = (in_port_t)htons(30000);
  name.sin_addr.s_addr = htonl(INADDR_ANY);
  bind(listener_d, (struct sockaddr *) &name, sizeof(name));   ← 把套接字绑定到30 000端口。

  listen(listener_d, 10);   ← 把监听队列长度设为10。
  puts("Waiting for connection");
  while (1) {   ← 你需要循环“接受连接，然后开始对话”这部分代码。
    struct sockaddr_storage client_addr;
    unsigned int address_size = sizeof(client_addr);
    int connect_d = accept(listener_d, (struct sockaddr *)&client_addr, &address_size);
    char *msg = advice[rand() % 5];   ← 接受来自客户端的连接。

    send(connect_d, msg, strlen(msg), 0);
    close(connect_d);   ← 开始和客户端通信。
  }
  return 0;
}
```

如果想拿附加分，就补上#include语句，让程序能够正确运行。但程序员好像忘了一件事，到底是什么事呢？提示：注意系统调用。

程序员忘了 检查是否发生错误。 ← 每次都必须检查socket、bind、listen、accept和send这些系统调用是否返回-1。



试驾

编译服务器，看看会发生什么。

```
File  Edit  Window  Help  I'mTheServer
> gcc advice_server.c -o advice_server
> ./advice_server
Waiting for connection
```

趁服务器还在运行，我们打开另一个控制台，用telnet连接几次试试。

```
File  Edit  Window  Help  I'mTelnet
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
One word: inappropriate
Connection closed by foreign host.
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
You might want to rethink that haircut
Connection closed by foreign host.
>
```

太好了，服务器正确运行了，你用127.0.0.1作为IP地址，因为客户端和服务器在同一台机器上运行。你也可以从其他地方连接服务器，我们将得到同样的答复。

服务器有时不能正常启动

如果我先启动服务器，然后运行客户端，没问题……

服务器控制台

```
File Edit Window Help I'mTheServer
> ./advice_server
Waiting for connection
```

服务器启动了。

客户端控制台

```
File Edit Window Help I'mTheClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
One word: inappropriate
Connection closed by foreign host.
>
```

服务器响应了。

……然后我关闭服务器又很快重启，客户端就再也得不到响应了！

服务器控制台

按Ctrl-C关闭服务器。

```
File Edit Window Help I'mTheServer
> ./advice_server
Waiting for connection
^C
> ./advice_server
Waiting for connection
```

服务器重启了。

客户端控制台

```
File Edit Window Help I'mTheClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
telnet: connect to address 127.0.0.1: Connection refused
telnet: Unable to connect to remote host
>
```

TMD？发生了什么事？

它没答？

第二次服务器“看起来”正确启动了，但客户端却无法得到响应，为什么会这样？

别忘了，这段代码没有检查错误，我们试着在代码中加一些错误检查的代码，看能不能弄清楚到底发生了什么。

妈妈说要检查错误

我们来检查下面这行代码的错误，它把套接字绑定到端口：

原来的代码

修改以后

~~bind (listener_d, (struct sockaddr *) &name, sizeof (name));~~

```
if (bind (listener_d, (struct sockaddr *) &name, sizeof(name)) == -1)
  error("无法绑定端口");
```

调用你不久之前写的error函数，它会显示错误原因然后退出程序。

再次关闭服务器，立即重启，这次得到了更多信息：

绑定失败！

```
File Edit Window Help I'mTheServer
> ./advice_server
Waiting for connection
^C
> ./advice_server
Can't bind the port: Address already in use
>
```

当服务器已经响应某个客户端时关闭服务器，然后立即重启，bind系统调用会失败。由于原来的代码没有检查错误，所以即使不能使用服务器端口，后面的代码还是会运行。

绑定端口有延时

当你在某个端口绑定了套接字，在接下来的30秒内，操作系统不允许任何程序再绑定它，包括上一次绑定这个端口的程序。只要在绑定前设置套接字的某个选项就可以解决这个问题。

一定要检查系统调用的错误。

需要用一个整型变量来保存选项。设为1，表示“重新使用端口”。

```
int reuse = 1;
if (setsockopt(listener_d, SOL_SOCKET, SO_REUSEADDR, (char *)&reuse, sizeof(int)) == -1)
  error("无法设置套接字的“重新使用端口”选项");
```

有了它，套接字就能重新使用端口。

通过以上代码，套接字就能重新使用已经绑定过的端口。也就是说你可以关闭服务器然后马上重启，在第二次绑定端口时就不会发生错误了。

从客户端读取数据

你已经会向客户端发消息了，那怎么从客户端读取数据呢？套接字用send()写数据，用**recv()**读数据：

<读了几个字节> = recv(<描述符>, <缓冲区>, <要读取几个字节>, 0);

如果用户在客户端输入一行文本，然后按下回车，recv()函数就会把文本保存在一个像这样的字符数组中：

W	h	o	'	s		t	h	e	r	e	?	\r	\n

recv()会返回14，因为客户端发送了14个字节。

牢记以下几点：

- **字符串不以\0结尾。**
- **当用户在telnet输入文本时，字符串以\r\n结尾。**
- **recv()将返回字符个数，如果发生错误就返回-1，如果客户端关闭了连接，就返回0。**
- **recv()调用不一定能一次接收到所有字符。**

最后一点很重要，它意味着可能需要多次调用recv()。

W	h	o	'	s		t	h	e	r	e	?	\r	\n

为了得到所有字符，可能需要多次调用recv()。

revc()用起来十分繁琐，最好把它封装在某个函数中，比如下面这个函数，它在指定数组中保存以\0结尾的字符串。

```
int read_in(int socket, char *buf, int len)
{
  char *s = buf;
  int slen = len;
  int c = recv(socket, s, slen, 0);
  while ((c > 0) && (s[c-1] != '\n')) {
    s += c; slen -= c;
    c = recv(socket, s, slen, 0);
  }
  if (c < 0)
    return c;
  else if (c == 0)
    buf[0] = '\0';
  else
    s[c-1]='\0';
  return len - slen;
}
```

这个函数读取\n前的所有字符。

循环读取字符，直到没有字符可读或读到了\n。

防止错误。

什么都没读到，返回一个空字符串。

用\0替换\r。

滑野雪

这是简化recv()的一种方法，你可以做得更好吗？为什么不自己写一个read_in()呢？我们在headfirstlabs.com等你的好消息。

代码熟食

这里还有一些在写服务器时会用到的代码，你能看懂它们是怎么工作的吗？

```
void error(char *msg)
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(1);
}
```

显示错误……

……然后停止程序。

你已经在本书中屡次使用了error函数

如果想让程序运行下去，就不要调用这个函数。

```
int open_listener_socket()
{
  int s = socket(PF_INET, SOCK_STREAM, 0);
  if (s == -1)
    error("Can't open socket");

  return s;
}
```

创建互联网流套接字。

是的，重新使用端口（这样重启服务器时就不会出错了）。

```
void bind_to_port(int socket, int port)
{
  struct sockaddr_in name;
  name.sin_family = PF_INET;
  name.sin_port = (in_port_t)htons(30000);
  name.sin_addr.s_addr = htonl(INADDR_ANY);
  int reuse = 1;
  if (setsockopt(socket, SOL_SOCKET, SO_REUSEADDR, (char *)&reuse, sizeof(int)) == -1)
    error("Can't set the reuse option on the socket");
  int c = bind (socket, (struct sockaddr *) &name, sizeof(name));
  if (c == -1)
    error("Can't bind to socket");
}
```

套接字名是互联网30 000端口

绑定30 000端口

```
int say(int socket, char *s)
{
  int result = send(socket, s, strlen(s), 0);
  if (result == -1)
    fprintf(stderr, "%s: %s\n", "和客户端通信时发生了错误", strerror(errno));
  return result;
}
```

向客户端发送字符串。

出错时别调用error()。你可不想因为一个客户端发生错误就关闭服务器。

下面就来试用一下这几个服务器函数……

练习

下面就开始写互联网knock-knock服务器的代码。这次你要写更多代码，不过可以使用上一页中现成的代码，我们为你开了一个头。

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <signal.h>
```

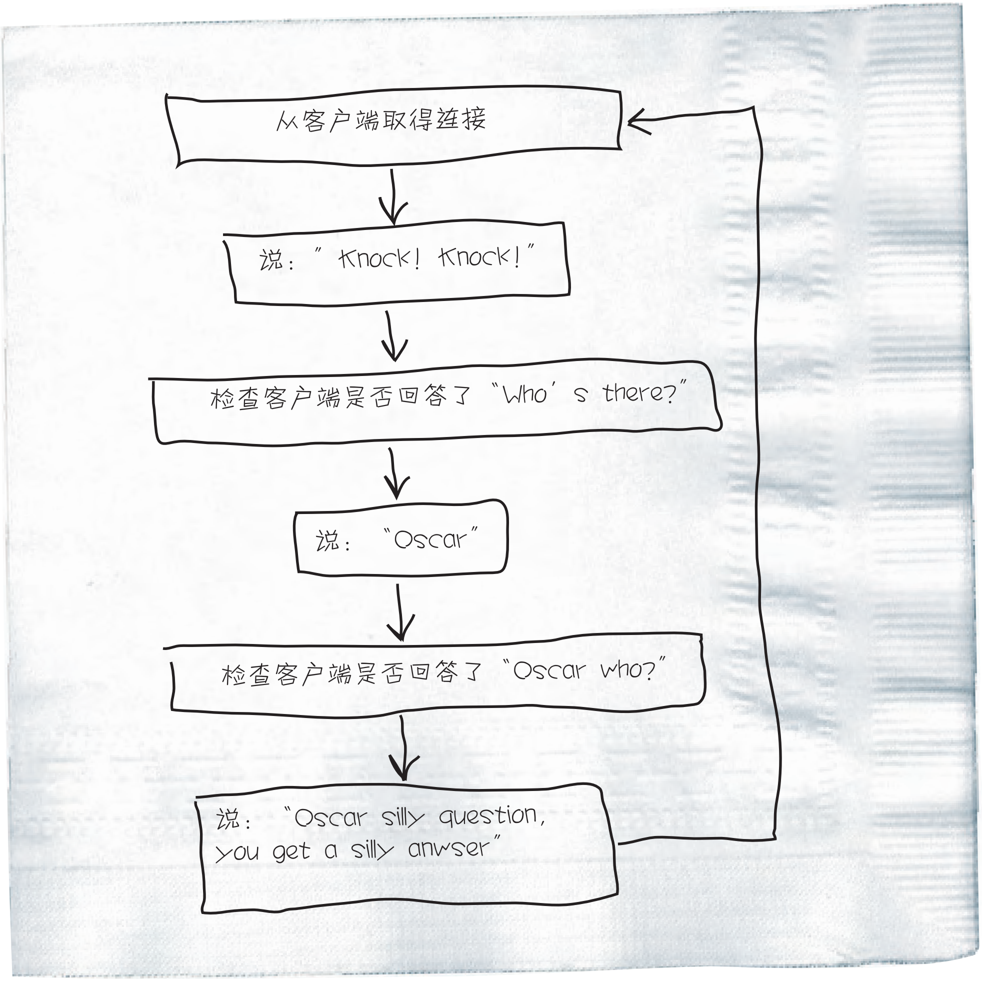
第479页的“代码熟食”放在这里。第453页的catch_signal()函数和第478页的read_in函数也要放在这里。

它将保存服务器的主监听套接字。

```
int listener_d;

void handle_shutdown(int sig)
{
  if (listener_d)
    close(listener_d);

  fprintf(stderr, "Bye!\n");
  exit(0);
}
```

如果有人在服务器运行期间按了Ctrl-C，这个函数就会赶在程序结束前关闭套接字。

主函数需要你来写。需要创建一个新的服务器套接字，然后保存在listener_d中；服务器套接字将绑定到30000端口；队列长度为10。程序流程图如下：



别忘了检查错误。如果用户回答错误就向它发送一条错误消息，然后关闭连接，等待其他客户端连接。**加油！**

练习解答

下面就开始写互联网knock-knock服务器的代码。这次你要写更多代码，不过可以使用第479页中现成的代码，我们为你开了一个头。

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <signal.h>
```

第479页的“代码熟食”放在这里。第453页的catch_signal()函数和第478页的read_in函数也要放在这里。

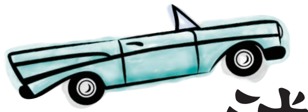
它将保存服务器的主监听套接字。

```
int listener_d;

void handle_shutdown(int sig)
{
  if (listener_d)
    close(listener_d);

  fprintf(stderr, "Bye!\n");
  exit(0);
}
```

如果有人在服务器运行期间按了Ctrl-C，这个函数就会赶在程序结束前关闭套接字。

你的代码应该看起来像下面这样，不一模一样也没关系，只要代码能按正确的套路说笑话并且能处理错误就行。

```
int main(int argc, char *argv[])
{
  if (catch_signal(SIGINT, handle_shutdown) == -1)
    error("Can't set the interrupt handler");      如果有人按了Ctrl-C就调用handle_shutdown()。
  listener_d = open_listener_socket();
  bind_to_port(listener_d, 30000);      在30000端口创建套接字。
  if (listen(listener_d, 10) == -1)      把队列长度设为10。
    error("Can't listen");
  struct sockaddr_storage client_addr;
  unsigned int address_size = sizeof(client_addr);
  puts("Waiting for connection");
  char buf[255];
  while (1) {      监听连接。
    int connect_d = accept(listener_d, (struct sockaddr *)&client_addr, &address_size);
    if (connect_d == -1)
      error("Can't open secondary socket");      向客户端发送数据。
    if (say(connect_d,
              "Internet Knock-Knock Protocol Server\r\nVersion 1.0\r\nKnock! Knock!\r\n>")
!= -1) {
      read_in(connect_d, buf, sizeof(buf));      从客户端读取数据。
      if (strncasecmp("Who's there?", buf, 12))
        say(connect_d, "You should say 'Who's there?'!");      检查用户的回答。
      else {
        if (say(connect_d, "Oscar\r\n> ") != -1) {
          read_in(connect_d, buf, sizeof(buf));
          if (strncasecmp("Oscar who?", buf, 10))
            say(connect_d, "You should say 'Oscar who?'!\r\n");
          else
            say(connect_d, "Oscar silly question, you get a silly answer\r\n");
        }
      }
    }      关闭我们用来对话的副套接字。
    close(connect_d);
  }
  return 0;
}
```



试驾

knock-knock服务器已经竣工，下面就来编译运行。

服务器控制台

```
File  Edit  Window  Help  I'mTheServer
> gcc ikkp_server.c -o ikkp_server
> ./ikkp_server
Waiting for connection
```

服务器正在等待连接。打开另一个控制台，用telnet连接它。

客户端控制台

```
File  Edit  Window  Help  I'mTheClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Internet Knock-Knock Protocol Server
Version 1.0
Knock! Knock!
> Who's there?
Oscar
> Oscar who?
Oscar silly question, you get a silly answer
Connection closed by foreign host.
```

服务器开始讲笑话了。如果违反协议，乱回答一句会怎么样?

客户端控制台

```
File  Edit  Window  Help  I'mTheClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Internet Knock-Knock Protocol Server
Version 1.0
Knock! Knock!
> Come in
You should say 'Who's there?'!Connection closed by foreign host.
>
```

服务器成功校验了你发送给它的数据，然后立马关闭了连接。当你不想运行服务器时，可以切回服务器窗口按Crtl-C关闭，它还会和你说拜拜：

服务器控制台

```
File  Edit  Window  Help  I'mTheServer
> gcc ikkp_server.c -o ikkp_server
> ./ikkp_server
Waiting for connection
^CBye!
>
```

太好了！服务器不辱使命。

真的吗?

一次只能服务一个人

服务器代码有一个问题。想象一下，如果有人连上了服务器，但他在回复时动作有些慢：

服务器运行在互联网的某台计算机上。

```
File Edit Window Help I'mTheClient
> telnet knockknockster.com 30000
Trying knockknockster.com...
Connected to localhost.
Escape character is '^]'.
Internet Knock-Knock Protocol Server
Version 1.0
Knock! Knock!
> Who's there?
Oscar
>
```

然后其他人就连不上服务器了，因为服务器还在服务前面那个人：

```
File Edit Window Help I'mAnotherClient
> telnet knockknockster.com 30000
Trying knockknockster.com...
Connected to localhost.
Escape character is '^]'.
```

问题出在服务器还在同第一个人通信，主服务器套接字会让客户端一直等下去，直到服务器再次调用accept()系统调用，但因为已经有人连接了，所以这个过程可能会有点长。

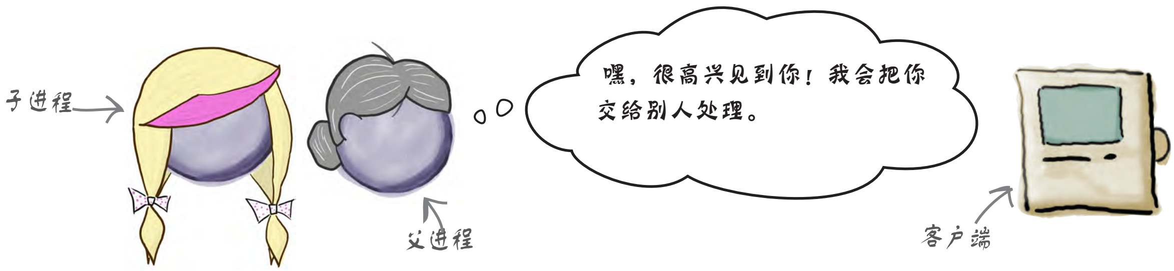
脑力风暴

服务器无法响应第二个用户，因为它正在处理与第一个用户之间的对话。你有没有学过什么方法可以同时处理两个客户端？

为每个客户端fork()一个子进程

客户端连到服务器以后会启用一个新创建的套接字对话，也就是说主服务器套接字可以去找下一个客户端，我们来试试。

当客户端连接时，可以用fork()克隆一个独立的子进程来处理它和服务器之间的对话。



当客户端在与子进程通信时，服务器的父进程可以继续连接下一个客户端。

父子进程使用不同套接字

有一件事你必须铭记于心，服务器的父进程只需要用主监听套接字（用来接受新的连接），而子进程只需要处理accpet()创建的副套接字。也就是说，父进程克隆出子进程后可以关闭副套接字，而子进程可以关闭主监听套接字。

克隆出子进程后，父进程就可以关闭这个套接字。

```
close(connect_d);
close(listener_d);
```

子进程创建以后就可以关闭这个套接字。

这里没有蠢问题

问： 如果为每个客户端都创建新进程，那么当有无数客户端连接服务器时计算机上岂不是会有无数进程？

答： 是的，如果你觉得会有很多客户端连接服务器，就需要控制创建进程的上限。子进程在处理完一个客户端后可以向你发出信号，利用这点就可以用有限的子进程处理无限的客户端。

磨笔上阵

我们修改了服务器的代码，现在它能克隆独立的子进程来和客户端通信……眼看就要完成了，你能找到漏掉的代码吗？

```
while (1) {
  int connect_d = accept(listener_d, (struct sockaddr *)&client_addr,
                          &address_size);
  if (connect_d == -1)
    error("Can't open secondary socket");

  if (......................) {
    close(......................);
    if (say(connect_d,
          "Internet Knock-Knock Protocol Server\r\nVersion 1.0\r\nKnock! Knock!\r\n> ")
        != -1) {
      read_in(connect_d, buf, sizeof(buf));

      if (strncasecmp("Who's there?", buf, 12))
        say(connect_d, "You should say 'Who's there?'!");
      else {
        if (say(connect_d, "Oscar\r\n> ") != -1) {
          read_in(connect_d, buf, sizeof(buf));

          if (strncasecmp("Oscar who?", buf, 10))
            say(connect_d, "You should say 'Oscar who?'!\r\n");
          else
            say(connect_d, "Oscar silly question, you get a silly answer\r\n");
        }
      }
    }
    close(......................);
    ......................   <—— 通信结束以后子进程应该干什么？
  }
  close(......................);
}
```

磨笔上阵 解答

我们已经修改了服务器的代码，现在它能克隆独立的子进程来和客户端通信……眼看就要完成，你将找出漏掉的代码。

```
while (1) {
  int connect_d = accept(listener_d, (struct sockaddr *)&client_addr,
                         &address_size);
  if (connect_d == -1)
    error("Can't open secondary socket");

  if ( !fork() ) {
    close( listener_d );
    if (say(connect_d,
           "Internet Knock-Knock Protocol Server\r\nVersion 1.0\r\nKnock! Knock!\r\n> ")
        != -1) {
      read_in(connect_d, buf, sizeof(buf));

      if (strncasecmp("Who's there?", buf, 12))
        say(connect_d, "You should say 'Who's there?'!");
      else {
        if (say(connect_d, "Oscar\r\n> ") != -1) {
          read_in(connect_d, buf, sizeof(buf));

          if (strncasecmp("Oscar who?", buf, 10))
            say(connect_d, "You should say 'Oscar who?'!\r\n");
          else
            say(connect_d, "Oscar silly question, you get a silly answer\r\n");
        }
      }
    }
    close( connect_d );
    exit(0);
  }
  close( connect_d );
}
```

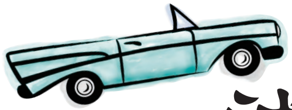
创建子进程，如果fork()调用返回0，就说明你在子进程中。

在子进程中，需要关闭主监听套接字。

子进程只用connect_d套接字和客户端通信。

一旦通信结束，子进程就可以关闭通信套接字了。

通信结束以后，子进程应该退出程序。这样就能防止子进程进入服务器的主循环。



试驾

试试修改后的服务器，你可以像刚刚一样编译运行：

服务器控制台

```
File Edit Window Help I'mTheServer
> gcc ikkp_server.c -o ikkp_server
> ./ikkp_server
Waiting for connection
```

打开另一个控制台，启动telnet，像刚才一样连接：

客户端控制台

```
File Edit Window Help I'mTheClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Internet Knock-Knock Protocol Server
Version 1.0
Knock! Knock!
> Who's there?
Oscar
>
```

看起来没什么区别，但只要你让客户端在笑话讲到一半的时候停在那里，就能看到修改后的效果：

假如你打开第三个控制台，就可以看到服务器现在有两个进程：一父一子：

unix和cygwin的ps命令可以显示当前正在运行的进程。

父进程

```
File Edit Window Help I'mJustCurious
> ps -a
 PID TTY          TIME CMD
14324 ttys002    0:00.00 ./ikkp_server
14412 ttys002    0:00.00 ./ikkp_server
>
```

子进程

即使第一个客户端还在和服务器通信，你仍然可以连接服务器：

另一个客户端控制台

```
File Edit Window Help I'mAnotherClient
> telnet 127.0.0.1 30000
Trying 127.0.0.1...
Connected to localhost.
Escape character is '^]'.
Internet Knock-Knock Protocol Server
Version 1.0
Knock! Knock!
>
```

你已经创建了一个互联网服务器，下面就来看看如何创建客户端，我们来写一个可以读取网页内容的程序。

自己动手写网络客户端

怎样才能写出自己的客户端程序？它和服务器之间的差别真的有那么大吗？为了体会两者的异同，下面就来写一个HTTP协议的网络客户端。

HTTP协议很像你之前写过的互联网knock-knock协议。协议是一段结构化对话，网络客户端和服务器必须谈得来才行。打开telnet，看看人家是怎么下载这个网页的：http://en.wikipedia.org/wiki/O’Reilly_Media。

动手吧！

绝大多数网络服务器运行在80端口。

维基百科的IP地址。试的时候可能得到一个不同的地址。

需要输入这两行。

然后需要按下回车输入空行。

服务器先回复了网页的一些附加信息。

```
File  Edit  Window  Help  I'mJustCurious
> telnet en.wikipedia.org 80
Trying 91.198.174.225...
Connected to wikipedia-lb.esams.wikimedia.org.
Escape character is '^]'.
GET /wiki/O'Reilly_Media http/1.1
Host: en.wikipedia.org

HTTP/1.0 200 OK
Server: Apache
...
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
"http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang=en" dir="ltr" class"client-nojs"
xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>O'Reilly Media - Wikipedia, the free encyclopedia</title>
...
```

在URL中，这个路径将跟在主机名后面。

在HTTP/1.1中，你需要说明要使用的主机名。

这是网页的HTML。

当程序连上网络服务器后，至少需要发送三样东西：

大多数网络客户端一般不止发送三条信息，但你只要发三条就行了。

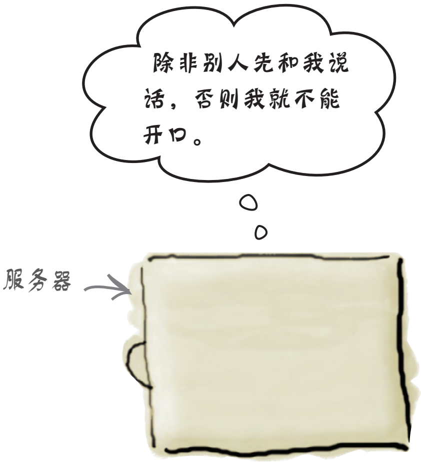
- **GET命令**
  `GET /wiki/O'Reilly_Media http/1.1`
- **主机名**
  主机：`en.wikipedia.org`
- **空行**

但你必须先连上服务器，然后才能向服务器发送数据。那怎么连接呢？

主动权在客户端手中

客户端和服务器使用套接字通信，但两者获取套接字的方式不同，服务器用BLAB四部曲取得套接字：

1. 绑定端口。
2. 监听。
3. 接受连接。
4. 开始通信。



服务器终其一生都在等待新客户端的连接。在客户端连接之前，它什么事都不能做。但客户端不一样，它想什么时候连接服务器并开始通信都可以。客户端只需两步就能取得套接字：

1. 连接远程端口。
2. 开始通信。

远程端口和IP地址

服务器在连接网络时必须决定使用哪个端口，而客户端除了要知道端口号还需要知道远程服务器的IP地址：

```
208.201.239.100
```

4个数字的IP地址是IPV4格式，它最终将被更长的IPV6地址取代。

IP地址难以记忆，所以人们一般使用域名。域名是一个好记的字符串，如：

www.oreilly.com

尽管人类喜欢用域名，但网络中的数据包只使用数字IP地址。

创建IP地址套接字

一旦客户端知道了服务器的地址和端口号，它就能创建客户端套接字了。客户端套接字和服务器套接字以相同的方式创建：

```
int s = socket(PF_INET, SOCK_STREAM, 0);
```

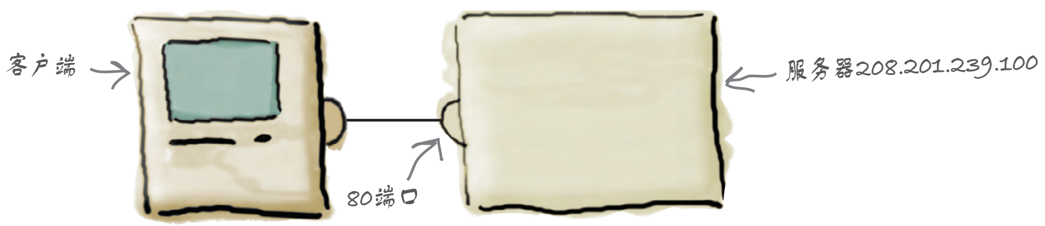
为了节约空间，我们在这个例子中没有检查错误，但你在自己的代码中必须那么做。

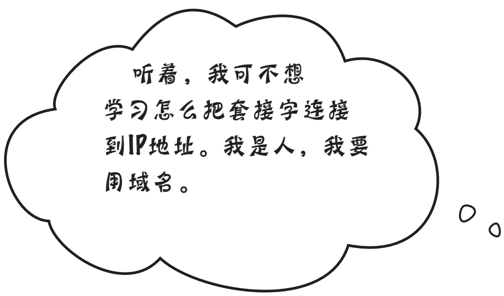
客户端和服务器处理套接字的方式不同，服务器会把套接字绑定到本地端口，而客户端会把套接字连接至远程端口：

这几行代码为208.201.239.100的80端口创建了一个套接字地址。

```
struct sockaddr_in si;
memset(&si, 0, sizeof(si));
si.sin_family = PF_INET;
si.sin_addr.s_addr = inet_addr("208.201.239.100");
si.sin_port = htons(80);
connect(s, (struct sockaddr *) &si, sizeof(si));
```

这行代码把套接字连接至远程端口。





以上代码适用于数字IP地址。

如果你想把套接字连接至远程域名，可以用getaddrinfo()函数。

getaddrinfo()获取域名的地址

域名系统（Domain Name System， DNS）是一本巨大的通讯录。计算机向网络发送数据包时需要在地址一栏填写数字形式的IP地址，而DNS可以把www.oreilly.com这样的域名转化为IP地址。

DNS是一本巨大的通讯录。

域名	地址
en.wikipedia.org	91.198.174.225
www.oreilly.com	208.201.239.100
www.oreilly.com	208.201.239.101

一些大网站有好几个IP地址。

计算机在创建网络数据包时要用到IP地址。

创建域名套接字

通常情况下，应该让客户端代码用DNS来创建套接字，这样用户就不需要自己去查找IP地址。为了使用DNS，需要以另一种方式构建客户端套接字：

```
#include <netdb.h>
...
struct addrinfo *res;
struct addrinfo hints;
memset(&hints, 0, sizeof(hints));
hints.ai_family = PF_UNSPEC;
hints.ai_socktype = SOCK_STREAM;
getaddrinfo("www.oreilly.com", "80", &hints, &res);
```

为了使用getaddrinfo()函数，需要包含这个头文件。

getaddrinfo()接收字符串格式的端口号。

创建www.oreilly.com地址80端口的名字资源。

getaddrinfo()会在堆上创建一种叫名字资源的新数据结构。给定域名和端口号，就可以得到名字资源。名字资源把计算机需要的IP地址隐藏了起来，大型网站通常有好几个IP地址，代码会从中挑选一个。随后便可以用名字资源创建套接字了。

现在就可以用名字资源来创建套接字了。

```
int s = socket(res->ai_family, res->ai_socktype,
                         res->ai_protocol);
```

最后，你可以连接远程套接字。因为名字资源在堆上创建，所以要用一个叫**freeaddrinfo()**的函数清除它。

```
connect(s, res->ai_addr, res->ai_addrlen);
freeaddrinfo(res);
```

res->ai_addr是远程主机加端口号的地址。

res->ai_addrlen是地址在存储器中的长度。

连接远程套接字。

连接以后可以用freeaddrinfo()函数删除地址数据。

一旦把套接字连接到远程端口，就可以用recv()和send()函数读写数据，你在服务器中已用过它们。你现在掌握的知识已经够写一个网络客户端了……

代码冰箱贴

网络客户端的代码如下，它将从维基百科下载某个页面的内容，然后在屏幕上显示。网址将通过参数传给程序。仔细思考一下，如果网络服务器使用了HTTP协议，你需要向它发送什么数据？

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>

void error(char *msg)
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(1);
}

int open_socket(char *host, char *port)
{
  struct addrinfo *res;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = PF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  if (getaddrinfo(host, port, &hints, &res) == -1)
    error("Can't resolve the address");
  int d_sock = socket(res->ai_family, res->ai_socktype,
                      res->ai_protocol);
  if (d_sock == -1)
    error("Can't open socket");
  int c = connect(d_sock, res->ai_addr, res->ai_addrlen);
  freeaddrinfo(res);
  if (c == -1)
    error("Can't connect to socket");
  return d_sock;
}
```

```
int say(int socket, char *s)
{
  int result = send(socket, s, strlen(s), 0);
  if (result == -1)
    fprintf(stderr, "%s: %s\n", "Error talking to the server",
strerror(errno));
  return result;
}

int main(int argc, char *argv[])
{
  int d_sock;

  d_sock = ..........................................................;
  char buf[255];

  sprintf(buf, .......................................................... , argv[1]);
  say(d_sock, buf);

  say(d_sock, ..........................................................);
  char rec[256];
  int bytesRcvd = recv(d_sock, rec, 255, 0);
  while (bytesRcvd) {
    if (bytesRcvd == -1)
      error("Can't read from server");

    rec[bytesRcvd] = ..........................................................;
    printf("%s", rec);
    bytesRcvd = recv(d_sock, rec, 255, 0);
  }

  .......................................................... ;
  return 0;
}
```

'\0'

"\r\n"

"Host: en.wikipedia.org\r\n\r\n"

"GET /wiki/%s http/1.1\r\n"

open_socket("en.wikipedia.org", "80")

"Host: en.wikipedia.org\r\n"

close(d_sock)

代码冰箱贴解答

网络客户端的代码如下，它将从维基百科下载某个页面的内容，然后在屏幕上显示。网址将通过参数传给程序。仔细思考一下，如果网络服务器使用了HTTP协议，你需要向它发送什么数据？

```
#include <stdio.h>
#include <string.h>
#include <errno.h>
#include <stdlib.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <unistd.h>
#include <netdb.h>

void error(char *msg)
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(1);
}

int open_socket(char *host, char *port)
{
  struct addrinfo *res;
  struct addrinfo hints;
  memset(&hints, 0, sizeof(hints));
  hints.ai_family = PF_UNSPEC;
  hints.ai_socktype = SOCK_STREAM;
  if (getaddrinfo(host, port, &hints, &res) == -1)
    error("Can't resolve the address");
  int d_sock = socket(res->ai_family, res->ai_socktype,
                      res->ai_protocol);
  if (d_sock == -1)
    error("Can't open socket");
  int c = connect(d_sock, res->ai_addr, res->ai_addrlen);
  freeaddrinfo(res);
  if (c == -1)
    error("Can't connect to socket");
  return d_sock;
}
```

```
int say(int socket, char *s)
{
  int result = send(socket, s, strlen(s), 0);
  if (result == -1)
    fprintf(stderr, "%s: %s\n", "Error talking to the server",
strerror(errno));
  return result;
}

int main(int argc, char *argv[])
{
  int d_sock;

  d_sock = open_socket("en.wikipedia.org", "80");
  char buf[255];

  sprintf(buf, "GET /wiki/%s http/1.1\r\n", argv[1]);
  say(d_sock, buf);

  say(d_sock, "Host: en.wikipedia.org\r\n\r\n");
  char rec[256];
  int bytesRcvd = recv(d_sock, rec, 255, 0);
  while (bytesRcvd) {
    if (bytesRcvd == -1)
      error("Can't read from server");

    rec[bytesRcvd] = '\0';
    printf("%s", rec);
    bytesRcvd = recv(d_sock, rec, 255, 0);
  }

  close(d_sock);
  return 0;
}
```

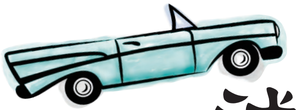
想要下载哪个网页，就根据它的网址创建字符串。

向主机发送数据和空行。

在字符数组的末尾加上\0使其成为字符串。

"\r\n"

"Host: en.wikipedia.org\r\n"



试驾

编译代码，运行网络客户端，它成功地从维基百科下载到了网页：

File Edit Window Help I'mTheWebClient

```
> gcc wiki_client.c -o wiki_client
> ./wiki_client "O'Reilly_Media"
HTTP/1.0 200 OK
Date: Fri, 06 Jan 2012 20:30:15 GMT
Server: Apache
...
Connection: close
<!DOCTYPE html PUBLIC "-//W3C//DTD XHTML 1.0 Transitional//EN"
 "http://www.w3.org/TR/xhtml1/DTD/xhtml1-transitional.dtd">
<html lang="en" dir="ltr" class="client-nojs" xmlns="http://www.w3.org/1999/xhtml">
<head>
<title>O'Reilly Media - Wikipedia, the free encyclopedia</title>
<meta http-equiv="Content-Type" content="text/html; charset=UTF-8" />
...
```

必须把所有空格都换成下划线(_)。

一开始你会得到应答头，它告诉了你一些关于服务器和网页的信息。

然后你从维基百科得到了网页的内容。

成功了！

客户端从命令行读取了网页的名字，然后连接到维基百科下载了网页。因为网页名要建立文件路径，所以必须用下划线（_）替换其中的空格。

滑野雪

为什么不让代码自动把空格替换成下划线？如何替换字符生成网址？详情请见：

http://www.w3schools.com/tags/ref_urlencode.asp

这里没有蠢问题

问： 我应该用IP地址还是域名创建套接字？

答： 最好用域名。一来域名比较好记，二来服务器有时会改变IP地址，但域名一般不会变。

问： 那我还用知道怎么连接IP地址吗？

答： 需要。如果你要连接的服务器没有在域名系统中注册，比如家庭网络中的计算机，你就需要知道如何用IP连接。

问： 我可以把IP地址作为`getaddrinfo()`的参数吗？

答： 可以。但如果你要连接IP地址，可以用第一版创建客户端套接字的代码，它更简单。

要点

- 协议是一段结构化对话。
- 服务器连接本地端口。
- 客户端连接远程端口。
- 客户端和服务器使用套接字通信。
- 用`send()`向套接字写数据。
- 用`recv()`从套接字读数据。
- HTTP是一种网络协议。

C语言工具箱

你已经学完了第11章，现在你的工具箱又加入了网络与套接字。关于本书的提示工具条的完整列表，请见附录ii。

telnet是一个简易网络客户端。

用socket()函数创建套接字。

服务器BLAB四部曲：

B = bind()

L = listen()

A = accept()

B = 开始对话

用fork()克隆子进程，同时处理多个客户端。

DNS= 域名系统

getaddrinfo()根据域名找地址。

12 线程

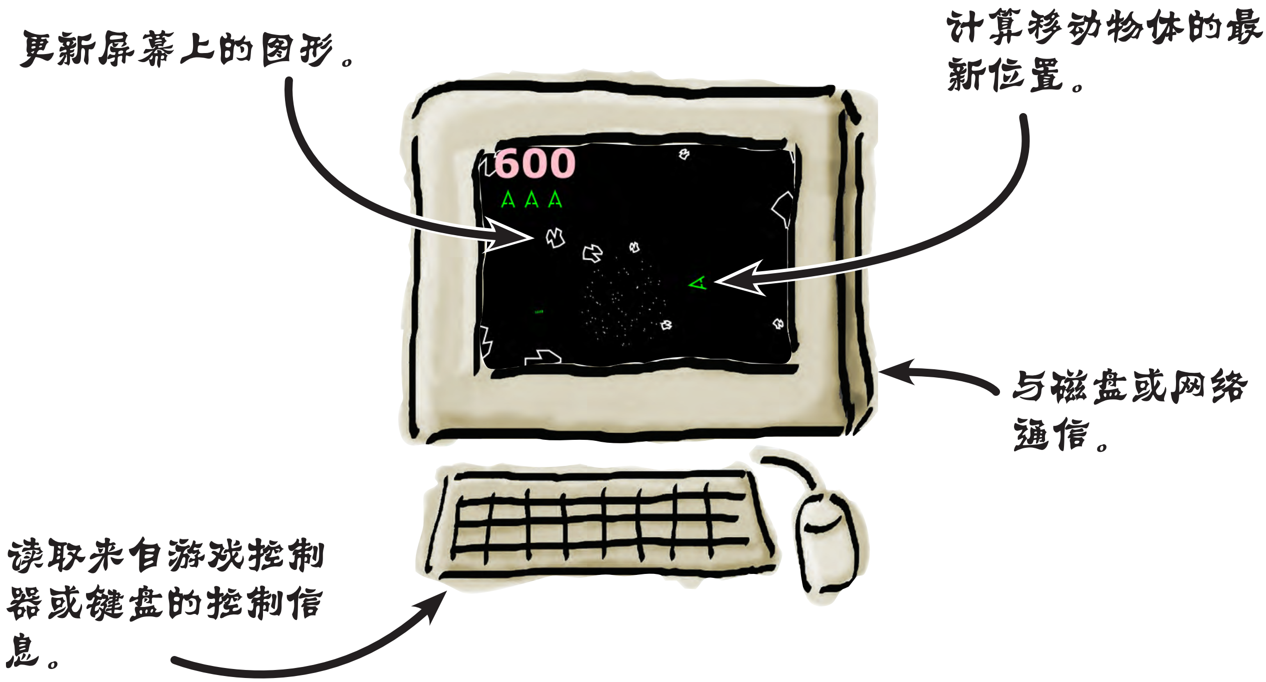
平行世界

程序经常需要同时做几件事。

POSIX线程可以派生几段并行执行的代码，从而提高代码的响应速度。但是要小心！线程虽然很强大，但它们之间可能发生冲突。本章你将学习如何用红绿灯来防止代码发生车祸。最终你将学会创建POSIX线程，并使用同步机制来保护共享数据的安全。

任务是串行的……还是……

想象你在用C语言写一个很复杂的程序，比如游戏，代码需要执行以下几个任务。



你的代码不但需要做这些事，而且需要同时做，其他程序也是如此。聊天程序需要一边从网络读取数据一边向网络发送数据；媒体播放器需要一边向显示器传送视频流一边监视来自用户控件的输入。

如何在代码中同时执行几个不同的任务?

……进程不是唯一答案

你已经学会了怎样让计算机同时做几件事：用**进程**。你在上一章中建立了一个网络服务器，它可以同时与几个不同客户端打交道。每当一个新用户连接时，服务器都会新建一个进程来处理新的会话。

难道每当想要同时做几件事时都得创建进程吗？不见得，有以下几个原因。

创建进程要花时间

有的机器新建进程只要花一丁点时间。虽然时间很短，但还是需要时间。如果你想要执行的任务才用几十毫秒，每次都创建进程就很低效。

共享数据不方便

当创建子进程时，子进程会自动包含父进程所有数据的副本。但这些只是副本，如果子进程想把数据发回父进程，就需要借助管道之类的东西。

进程真的很难

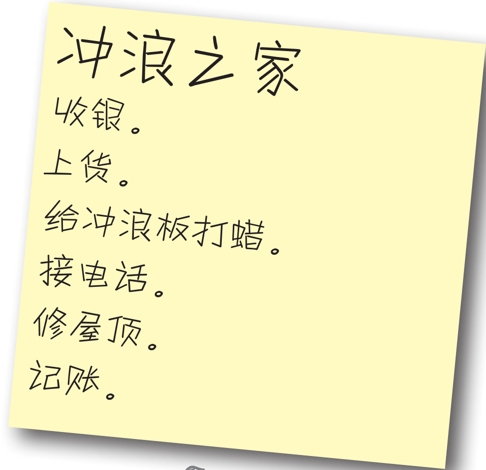
创建进程需要写很多代码，这会让代码又乱又长。

需要一个既可以快速启动任务，又可以共享现有数据，而且不需要写很多代码的东西。

你需要线程。

普通进程一次只做一件事

假设你有一张任务清单，上面列出了要做的事情：

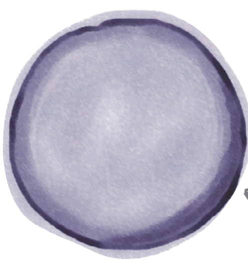


你也可以直接去冲浪。

我不能同时做这些事，你以为我是谁啊？

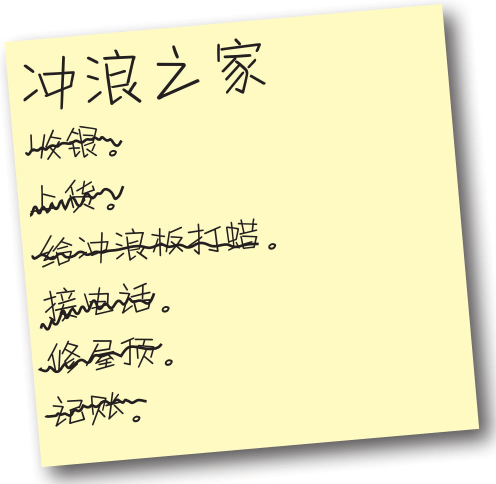
你没有办法同时做这些事情。如果顾客上门，需要放下手中上到一半的货，去招呼客人；如果下雨，就不能继续记账，得修一下屋顶；如果独自在店里干活，你就像一个进程，每次只做一件事。当然也可以不停切换任务，保持每件事都能推进下去，但如果这些任务中有一个是阻塞操作怎么办？假如你正在为顾客结账，电话响了怎么办？

到目前为止，你写过的所有程序都是单线程，这就好比进程中只有一个人在干活。

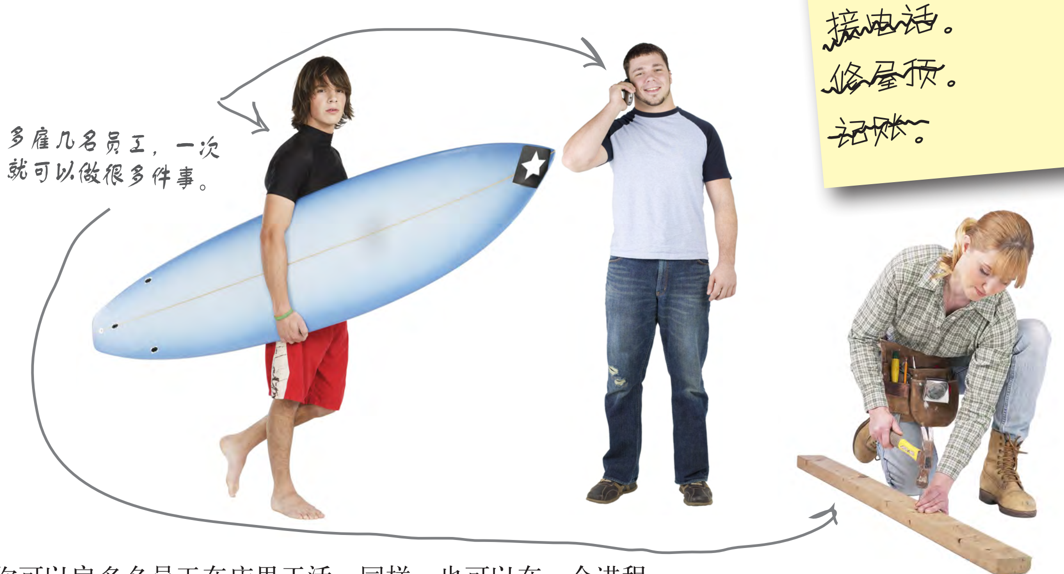


进程。

多雇几名员工：使用线程

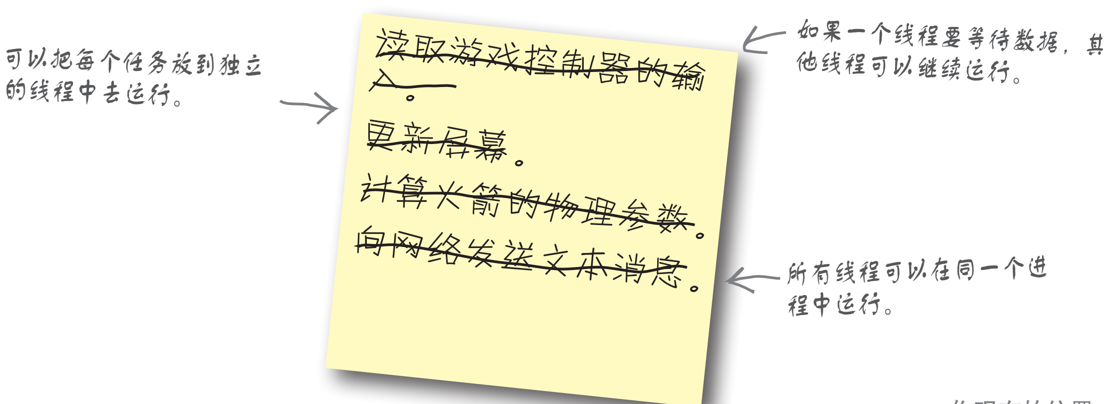


多线程程序就像有多名员工在店里工作：当一名员工在结账时，另一名员工可以给货架上货；当一名员工在给冲浪板打蜡时，其他员工可以干自己的活，完全不受干扰；当一个人在接电话时也不会打断店里其他人。



你可以雇多名员工在店里干活，同样，也可以在一个进程中使用多个线程。所有线程能访问同一段堆存储器，读写同一个文件，使用同一个网络套接字进行通信。当一个线程修改了某个全局变量，其他线程马上就能看到。

也就是说，可以为每个线程都分配一个独立的任务，让这些线程同时执行。



如何创建线程

你可以使用很多线程库，这里我们将使用最流行的一种：POSIX线程库，也叫**pthread**。可以在Cygwin、Linux和Mac上使用pthread。

假设你想在独立的线程中运行这两个函数：

线程函数的返回类型为void*。

```
void* does_not(void *a)
{
  int i = 0;
  for (i = 0; i < 5; i++) {
    sleep(1);
    puts("Does not!");
  }
  return NULL;
}
```

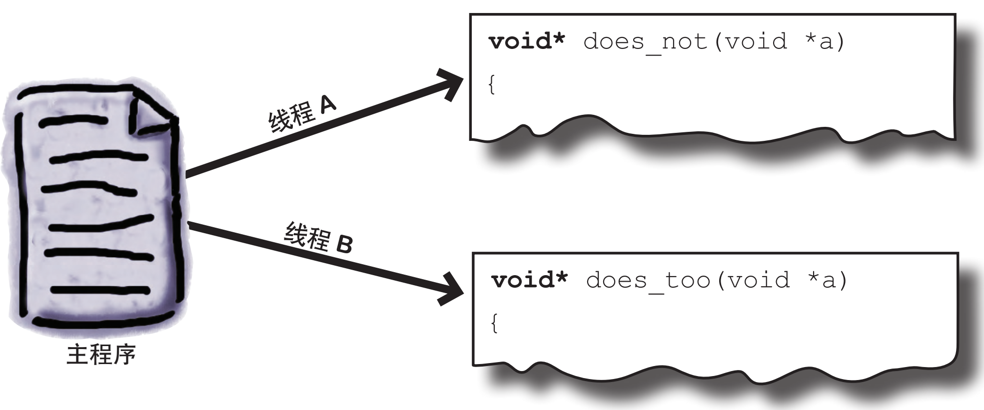
```
void* does_too(void *a)
{
  int i = 0;
  for (i = 0; i < 5; i++) {
    sleep(1);
    puts("Does too!");
  }
  return NULL;
}
```

没什么要返回的，那就返回NULL。

你发现了吗？两个函数都返回了void指针。

别忘了，void指针可以指向存储器中任何类型的数据，线程函数的返回类型必须是**void***。

你将在两个独立的线程中分别运行这两个函数。



你需要在两个独立的线程中并行地运行这两个函数，怎么才能做到呢？

用pthread_create创建线程

为了运行这两个函数，你需要进行一些设置，比如头文件和一个在程序出错时调用的error()函数。

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <errno.h>
#include <pthread.h>

void error(char *msg)
{
  fprintf(stderr, "%s: %s\n", msg, strerror(errno));
  exit(1);
}
```

代码的主要部分会用到这些头文件。

这是pthread库的头文库。

现在可以开始写主函数代码了。你将创建两个线程，每个线程都需要把信息保存在一个叫**pthread_t**的数据结构中，然后就可以用**pthread_create()**创建并运行线程。

```
pthread_t t0;
pthread_t t1;
if (pthread_create(&t0, NULL, does_not, NULL) == -1)
  error("无法创建线程t0");
if (pthread_create(&t1, NULL, does_too, NULL) == -1)
  error("无法创建线程t1");
```

它保存了线程的所有信息。

does_not是线程将运行的函数名。

创建线程。

每次都应该检查错误。

&t1是用来保存线程信息的数据结构的地址。

代码将以独立线程运行这两个函数。还没完，如果程序运行完这段代码就结束了，线程也会随之灭亡，因此必须等待线程结束：

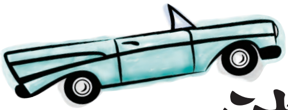
```
void* result;
if (pthread_join(t0, &result) == -1)
  error("无法回收线程t0");
if (pthread_join(t1, &result) == -1)
  error("无法收回线程t1");
```

函数返回的void指针会保存在这里。

pthread_join()函数会等待线程结束。

pthread_join()会接收线程函数的返回值，并把它保存在一个void指针变量中。一旦两个线程都结束了，程序就可以顺利退出了。

看看程序能否运行。



试驾

为了使用pthread库，必须在编译程序时链接它：

链接pthread库。

你的程序。

```
File Edit Window Help Don'tLoseTheThread
> gcc argument.c -lpthread -o argument
```

运行程序时，将看到两个函数同时运行：

当运行程序时，消息的顺序可能不同。

```
File Edit Window Help Don'tLoseTheThread
> ./argument
Does too!
Does not!
Does too!
Does not!
Does too!
Does not!
Does too!
Does not!
Does not!
Does too!
>
```

这里没有蠢问题

问： 既然两个函数同时运行，为什么字母没有混在一起，而是一行一条消息？

答： 因为标准输出就是那样工作的，puts()会一次输出整条字符串。

问： 我去掉了**sleep()**函数，为什么程序先显示一个函数的所有输出，然后再显示另一个函数的所有输出？

答： 在不调用sleep()的情况下，大多数计算机会很快地运行完代码，第一个函数将在第二个函数开始运行之前就结束。

啤酒冰箱贴

派对开始了，倒计数啤酒瓶数。下面这段代码运行了20个线程，总共有200万瓶啤酒。看看你能否找到丢失的代码，搞定以后干杯庆祝一下。

```
int beers = 2000000;
void* drink_lots(void *a)
{
  int i;
  for (i = 0; i < 100000; i++) {
    beers = beers - 1;
  }
  return NULL;
}
int main()
{
  pthread_t threads[20];
  int t;
  printf("%i bottles of beer on the wall\n%i bottles of beer\n", beers, beers);
  for (t = 0; t < 20; t++) {

    ............................(............................, NULL, ............................, NULL);
  }
  void* result;
  for (t = 0; t < 20; t++) {

    ............................(threads[t], &result);
  }
  printf("There are now %i bottles of beer on the wall\n", beers);
  return 0;
}
```

一开始有200万瓶啤酒。

每个线程都会运行这个函数。

函数会把beers变量的值减去10万。

将创建20个线程来运行这个函数。

为了节约纸张，这个例子跳过了错误检查，但你可别这么做！

代码会等待所有线程结束。

pthread_join　**threads**　**threads[t]**

pthread_create　**&threads[t]**　**drink_lots**

啤酒冰箱贴解答

派对开始了，倒计数啤酒瓶数。下面这段代码运行了20个线程，总共有200万瓶啤酒。请找到丢失的代码。

```
int beers = 2000000;
void* drink_lots(void *a)
{
  int i;
  for (i = 0; i < 100000; i++) {
    beers = beers - 1;
  }
  return NULL;
}
int main()
{
  pthread_t threads[20];
  int t;
  printf("%i bottles of beer on the wall\n%i bottles of beer\n", beers, beers);
  for (t = 0; t < 20; t++) {
    pthread_create(&threads[t], NULL, drink_lots, NULL);
  }
  void* result;
  for (t = 0; t < 20; t++) {
    pthread_join(threads[t], &result);
  }
  printf("There are now %i bottles of beer on the wall\n", beers);
  return 0;
}
```

为了节约纸张，我们跳过了错误检查，但你可别那么做！

threads

threads[t]

试驾

仔细观察刚才那个程序，当多次运行程序时会发生：

```
File Edit Window Help Don'tLoseTheThread
> ./beer
2000000 bottles of beer on the wall
2000000 bottles of beer
There are now 0 bottles of beer on the wall
> ./beer
2000000 bottles of beer on the wall
2000000 bottles of beer
There are now 883988 bottles of beer on the wall
> ./beer
2000000 bottles of beer on the wall
2000000 bottles of beer
There are now 945170 bottles of beer on the wall
>
```

20个线程把beers变量减为了0。

嘿，等等……

TMD，怎么回事？

甜蜜的

大多数情况下，代码没有把beers变量减为0。

奇怪，beers变量的初始值是200万，每个线程都把它的值减去10万，一共有20个线程，beers变量不应该每次都减到0吗？

脑力风暴

再次检查代码。试想当多个线程在同一刻运行时会发生什么？为什么结果不可预测？为什么所有线程运行过后beers变量没有减到0？把答案写在下面。

线程不安全

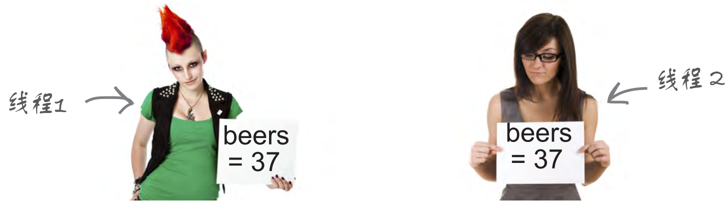
线程最大的优点在于很多不同任务可以同时运行，并访问相同数据，而不同任务可以访问相同数据恰恰也是线程的缺点……

不像第一个程序，第二个程序的线程读取并修改了存储器中的共享数据：beers变量。这有什么问题吗？好，我们来看一下当两个线程试图用以下代码减小beers值时会发生什么：

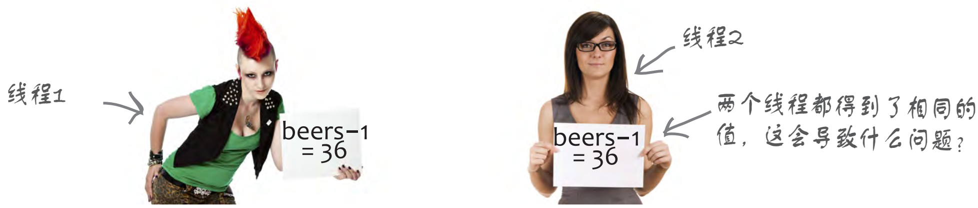
```
beers = beers - 1;
```

想象两个线程在同一时刻运行这行代码。

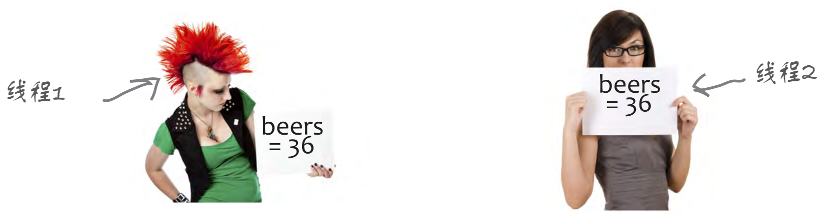
1 首先，两个线程都需要读取当前beers变量的值。



2 接着每个线程都将数字减1。



3 最后每个线程都将beers-1这个值写回beers变量。



两个线程都想把beers值减1，却没有成功。两个线程只把beers值减去了1，而不是2，这就是为什么beers变量没有减到0，因为线程之间会相互影响。

为什么结果是不可预测的呢？因为线程每次运行这行代码的顺序都不一样。线程有时不会撞车，有时则会撞得车毁人亡。

小心提防那些非线程安全的代码。

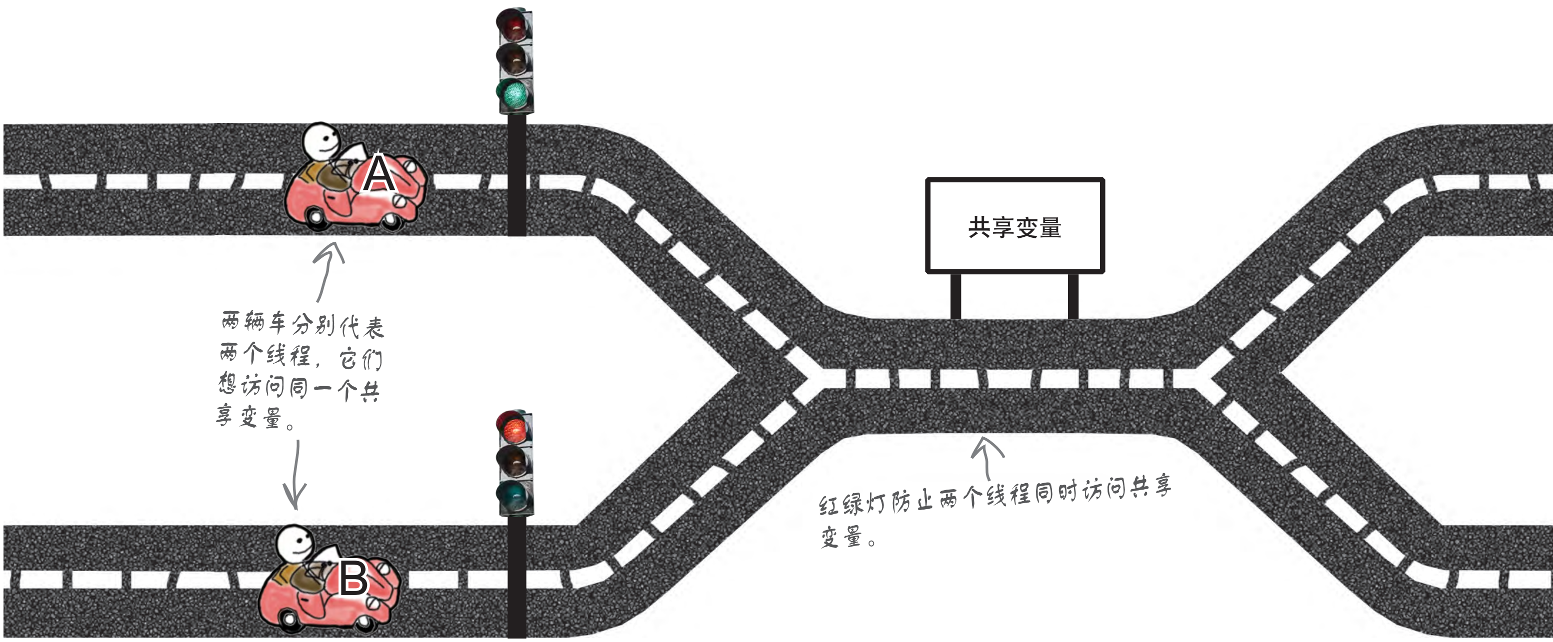
怎么才能知道一段代码是否线程安全呢？通常当两个线程读写相同变量时，代码就是非线程安全的。

增设红绿灯

多线程程序很强大，同时它们的行为也不可预测，除非采取一些控制手段。

假设两辆车想要驶过一段羊肠小道。为了防止交通事故，你可以增设红绿灯，它可以防止两辆车同时访问共享资源。

如果想防止两个或多个线程访问共享数据资源，也可以采取相同的方法：增设红绿灯。这样两个线程就不能同时读取相同数据，并把它写回。



用来防止线程发生车祸的红绿灯就叫互斥锁，它们是把代码变为线程安全最简单的方法。

有时也叫锁。

互斥就是相互排斥的意思。

用互斥锁来管理交通

为了保护某段代码的安全，你需要创建互斥锁：

```
pthread_mutex_t a_lock = PTHREAD_MUTEX_INITIALIZER;
```

互斥锁必须对所有可能发生冲突的线程可见，也就是说它是一个全局变量。

PTHREAD_MUTEX_INITIALIZER实际上是一个宏，当编译器看到它，就会插入创建互斥锁的代码。

1. **红灯停。**
你需要把第一盏红绿灯放在这段代码的开头，pthread_mutex_lock()只允许一个线程通过，其他线程运行到这行代码时必须等待。

```
pthread_mutex_lock(&a_lock);
/* 含有共享数据的代码从这里开始 */
```

一次只有一个线程能通过这里。

2. **绿灯行。**
当线程到达代码的尾部就会调用pthread_mutex_unlock()把红绿灯调回绿灯，其他线程就能进入这段代码了：

```
/* ...代码结束了 */
pthread_mutex_unlock(&a_lock);
```

既然你知道了怎么在代码中创建锁，也就能精确控制线程的行为了。

把long值传给线程函数

线程函数可以接收一个void指针作为参数，并返回一个void指针值。通常你希望把某个整型值传给线程，并让它返回某个整型值，一种方法是用**long**，因为它的大小和void指针相同，可以把它保存在void指针变量中。

```
void* do_stuff(void* param)
{
  long thread_no = (long)param;
  printf("Thread number %ld\n", thread_no);
  return (void*)(thread_no + 1);
}

int main()
{
  pthread_t threads[3];
  long t;
  for (t = 0; t < 3; t++) {
    pthread_create(&threads[t], NULL, do_stuff, (void*)t);
  }
  void* result;
  for (t = 0; t < 3; t++) {
    pthread_join(threads[t], &result);
    printf("Thread %ld returned %ld\n", t, (long)result);
  }
  return 0;
}
```

线程函数可以接收一个void指针类型的参数。

把它转回long。

返回时将其类型转化为void指针。

将long型变量t的值转化为void指针类型。

在使用前先把它转化为long。

```
File Edit Window Help Don'tLoseTheThread
> ./param_test
Thread number 0
Thread 0 returned 1
Thread number 1
Thread number 2
Thread 1 returned 2
Thread 2 returned 3
>
```

每个线程都接收一个数字作为线程号。

每个线程都返回线程号+1。

练习

找到上锁的位置实非易事，而锁的位置会改变代码的运行方式。下面有两个不同版本的drink_lots()函数，它们以不同方式为代码上了锁。

版本一

```
pthread_mutex_t beers_lock = PTHREAD_MUTEX_INITIALIZER;
void* drink_lots(void *a)
{
  int i;
  pthread_mutex_lock(&beers_lock);
  for (i = 0; i < 100000; i++) {
    beers = beers - 1;
  }
  pthread_mutex_unlock(&beers_lock);
  printf("beers = %i\n", beers);
  return NULL;
}
```

版本二

```
pthread_mutex_t beers_lock = PTHREAD_MUTEX_INITIALIZER;
void* drink_lots(void *a)
{
  int i;
  for (i = 0; i < 100000; i++) {
    pthread_mutex_lock(&beers_lock);
    beers = beers - 1;
    pthread_mutex_unlock(&beers_lock);
  }
  printf("beers = %i\n", beers);
  return NULL;
}
```

两段代码都用互斥锁来保护beers变量的安全，并在退出前显示了beers值。由于它们在不同的位置使用了锁，因此在屏幕上输出了不同结果。

你能弄清哪段代码对应哪个版本吗？

```
File Edit Window Help Don'tLoseTheThread
> ./beer_fixed_strategy_1
2000000 bottles of beer on the wall
2000000 bottles of beer
beers = 1900000
beers = 1800000
beers = 1700000
beers = 1600000
beers = 1500000
beers = 1400000
beers = 1300000
beers = 1200000
beers = 1100000
beers = 1000000
beers = 900000
beers = 800000
beers = 700000
beers = 600000
beers = 500000
beers = 400000
beers = 300000
beers = 200000
beers = 100000
beers = 0
There are now 0 bottles of beer on the wall
>
```

找到对应输出的代码。

```
File Edit Window Help Don'tLoseTheThread
> ./beer_fixed_strategy_2
2000000 bottles of beer on the wall
2000000 bottles of beer
beers = 63082
beers = 123
beers = 104
beers = 102
beers = 96
beers = 75
beers = 67
beers = 66
beers = 65
beers = 62
beers = 58
beers = 56
beers = 51
beers = 41
beers = 36
beers = 30
beers = 28
beers = 15
beers = 14
beers = 0
There are now 0 bottles of beer on the wall
>
```

练习解答

找到上锁的位置实非易事，而锁的位置会改变代码的运行方式。下面有两个不同版本的drink_lots()函数，它们以不同方式为代码上了锁。

版本一

```
pthread_mutex_t beers_lock = PTHREAD_MUTEX_INITIALIZER;
void* drink_lots(void *a)
{
  int i;
  pthread_mutex_lock(&beers_lock);
  for (i = 0; i < 100000; i++) {
    beers = beers - 1;
  }
  pthread_mutex_unlock(&beers_lock);
  printf("beers = %i\n", beers);
  return NULL;
}
```

版本二

```
pthread_mutex_t beers_lock = PTHREAD_MUTEX_INITIALIZER;
void* drink_lots(void *a)
{
  int i;
  for (i = 0; i < 100000; i++) {
    pthread_mutex_lock(&beers_lock);
    beers = beers - 1;
    pthread_mutex_unlock(&beers_lock);
  }
  printf("beers = %i\n", beers);
  return NULL;
}
```

两段代码都用互斥锁来保护beers变量的安全，并在退出前显示了beers值。由于它们在不同的位置使用了锁，因此在屏幕上输出了不同结果。

请弄清哪段代码对应哪个版本。

```
File Edit Window Help Don'tLoseTheThread
> ./beer_fixed_strategy_1
2000000 bottles of beer on the wall
2000000 bottles of beer
beers = 1900000
beers = 1800000
beers = 1700000
beers = 1600000
beers = 1500000
beers = 1400000
beers = 1300000
beers = 1200000
beers = 1100000
beers = 1000000
beers = 900000
beers = 800000
beers = 700000
beers = 600000
beers = 500000
beers = 400000
beers = 300000
beers = 200000
beers = 100000
beers = 0
There are now 0 bottles of beer on the wall
>
```

找到对应输出的代码。

```
File Edit Window Help Don'tLoseTheThread
> ./beer_fixed_strategy_2
2000000 bottles of beer on the wall
2000000 bottles of beer
beers = 63082
beers = 123
beers = 104
beers = 102
beers = 96
beers = 75
beers = 67
beers = 66
beers = 65
beers = 62
beers = 58
beers = 56
beers = 51
beers = 41
beers = 36
beers = 30
beers = 28
beers = 15
beers = 14
beers = 0
There are now 0 bottles of beer on the wall
>
```

恭喜！你已经（快要）看完这本书了。打开一瓶啤酒，庆祝一下吧！

是时候决定你将成为哪种类型的C程序员了。写纯C代码的Linux黑客？还是为Arduino那种小装置写嵌入式C的匠人？或是转而成为一名使用C++的游戏开发人员？或使用Objective-C的Mac及iOS程序员？

无论你的选择是什么，你都已经成为了C社区的一份子。在这里，你们使用同一种语言，并深深热爱着它。这种语言创建的软件比其他任何语言都要多；它是整个互联网和几乎所有操作系统的基础；几乎所有其他语言都是用它写的；几乎所有电子设备的处理器都可以用它来编程，大到飞机卫星，小到手表手机。

欢迎你！一年级C黑客！

这里没有蠢问题

问：为了支持多线程，我的计算机必须有多个处理器吗？

答： 不必。绝大多数计算机都使用多核处理器。也就是说CPU中有一些小型处理器，它们可以一次做几件事情。即便代码运行在一台单核/单处理器的计算机上，也还是能运行多线程程序。

问：怎么运行？

答： 操作系统会在多个线程之间快速地切换，看起来就好像在同时做多件事。

问：线程能让程序变得更快吗？

答： 也不一定，尽管线程可以帮助你利用机器上更多的处理器和核，但你还是需要控制代码中锁的数量，如果用了太多锁，代码可能会像单线程程序一样慢。

问：怎样设计高效的多线程程序？

答： 减少线程需要访问的共享数据的数量。如果线程无需访问很多共享数据，那么多个线程等一个线程的情况就很少出现，速度会大大提高。

问：线程要比进程快？

答： 通常是这样，因为创建进程要比创建线程花更多时间。

问：听说互斥锁会引发“死锁”，那是什么玩意儿？

答： 假设你有两个线程，它们都想得到互斥锁A和B。倘若第一个线程得到了A，第二个线程得到了B，这两个线程就会陷入死锁。因为第一个线程无法得到B，第二个线程无法得到A，它俩都停滞不前。

C语言工具箱

你已经学完了第12章，现在你的工具箱又加入了线程。关于本书的提示工具条的完整列表，请见附录ii。

有了线程，进程一次就能做多件事。

普通进程一次只做一件事。

线程是“轻量级进程”。

POSIX线程（pthread）是一个线程库。

pthread_create()创建线程来运行函数。

pthread_join()会等待线程结束。

线程共享相同的全局变量。

如果线程读取并更新了相同变量，代码的运行结果将不可预测。

pthread_mutex_lock()在代码中创建互斥锁。

互斥锁是用来保护共享数据的锁。

pthread_mutex_unlock()释放互斥锁。

实验人：　　　　　　实验日期：

C语言实验室 3

爆破彗星

本实验会给你一份说明书，它描述了一个程序，你需要运用你在前几章中学到的知识构建这个程序。

这个项目比你之前见识到的项目都要大，所以动手之前请阅读完全部内容，并给自己一点时间。不要担心会被难倒，这里没有新概念，你也可以接着往后读，回过头再来做这个实验。

我们还为你补充了一些设计上的细节，万事俱备，包你能写出代码。

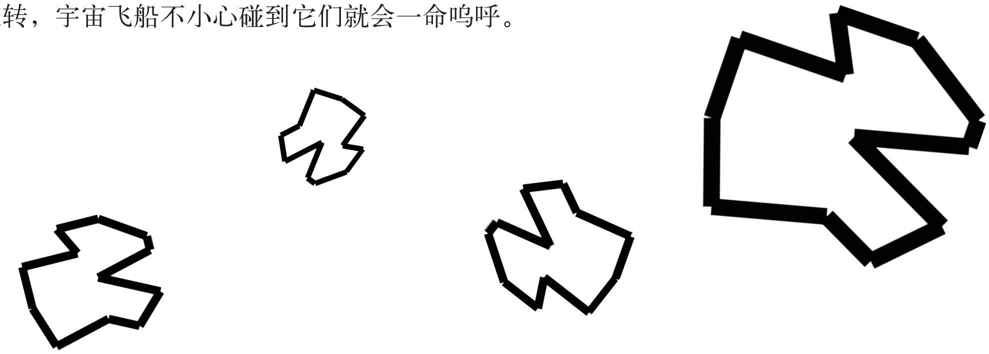
但需要你去实现程序，我们不会提供任何代码或答案。

经典街机游戏——爆破彗星

很多人学C语言是为了写游戏，在本实验中，你将向史上最受欢迎、最长寿的电子游戏——《爆破彗星》——致敬！

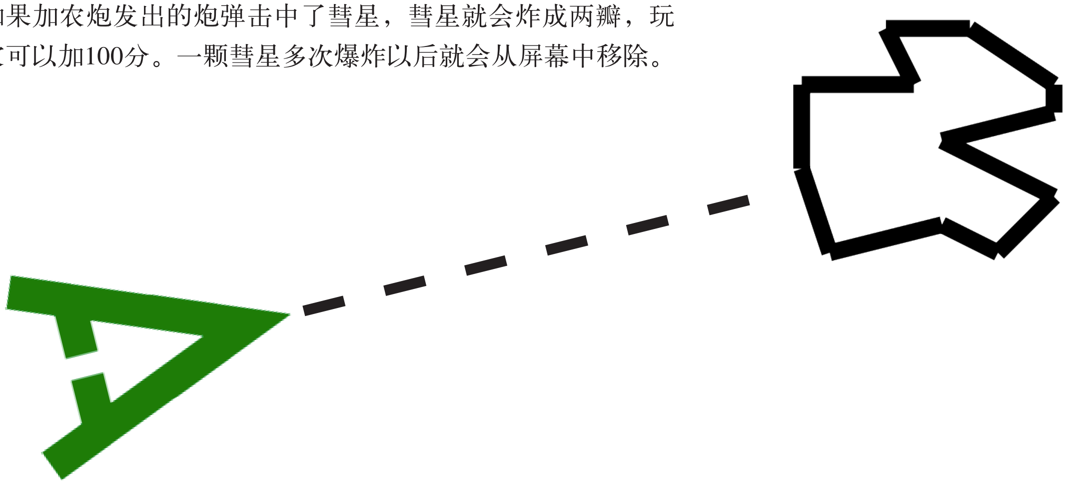
任务：闪避彗星并向它们开炮

彗星是你在游戏中需要消灭的敌人，它们在屏幕中缓缓地漂浮、旋转，宇宙飞船不小心碰到它们就会一命呜呼。



欢迎登上“向量号”宇宙飞船！你将用键盘控制飞船在屏幕中移动。飞船装备了加农炮，你可以向面前的彗星开炮。

如果加农炮发出的炮弹击中了彗星，彗星就会炸成两瓣，玩家可以加100分。一颗彗星多次爆炸以后就会从屏幕中移除。



如果飞船撞到彗星，就会丢一条命。你一共有三条命，用完游戏就结束了。

Allegro

Allegro是一款开源游戏开发库，用它创建的游戏代码可以在不同操作系统中编译运行。它支持Windows、Linux和Mac OS，甚至还有手机。

Allegro使用起来非常简单，麻雀虽小却五脏俱全。它能够处理声音、图形、动画和设备，如果你的计算机支持OpenGL，它还能处理三维图形。

OpenGL是一套与图形处理器交互的开放标准。你只要向OpenGL描述三维物体，它会替你处理大部分的数学问题。

安装Allegro

可以在Allegro Source Forge网站上下载Allegro的源代码：

http://alleg.sourceforge.net/projects/alleg

网站的更新速度比书快，这个URL可能会失效，何不用你喜欢的搜索引擎找一下？

可以从源代码仓库下载最新版本的代码，然后构建、安装Allegro。无论你用的是什么操作系统，都可以在网站上找到对应的安装教程。

你需要CMake

构建代码时你还需要安装一个叫`CMake`的工具。`CMake`是一个构建工具，它简化了在不同操作系统中构建C程序的工作。如果你要安装`CMake`，可以访问*http://www.cmake.org*。

我们在这个实验中所提供的代码仅适用于Allegro 5.0。

如果你下载安装的是更新的版本，可能要稍作修改。

Allegro能做什么？

Allegro库将为你处理：

- **GUI**

  Allegro将创建一个普通窗口来呈现你的游戏。这看起来没什么大不了，但不同操作系统创建窗口的方式天差地别，窗口与键盘鼠标交互的方式也不尽相同。

- **事件**

  每当按下一个键、移动一下鼠标或点击某个位置时，操作系统都会产生一个**事件**。事件其实就是一条数据，它告诉你计算机中发生了什么。事件在发送到程序之前会先进入一个队列。而Allegro简化了响应事件的过程，你能轻而易举地写出一段在用户按下空格（发射加农炮）时运行的代码。

- **定时器**

  你已经见识过了系统级定时器。Allegro提供了一种简单的方式为你的游戏加上“心跳”。游戏每秒钟会“心跳”好几次以确保显示能够持续更新。通过定时器，你就能创建一个按固定帧率（FPS）刷新屏幕的程序，比如每秒60帧。

- **图形缓冲**

  为了让你的游戏流畅运行，Allgero使用了双缓冲。双缓冲是一种游戏开发技术，它允许你先把图片缓存起来，然后再把它们显示到屏幕上，这样就能一次显示完整的一帧动画，游戏就更流畅了。

- **图形和变换**

  Allegro自带了一组图形原语，你可以用它们绘制直线、曲线、文本、实心图形和图片。如果你安装了OpenGL显卡驱动，还能绘制三维图形。除此之外，Allegro还支持变换，即在屏幕上旋转、平移、拉伸图形，这样你就能创建出逼真的宇宙飞船，并且让彗星在屏幕中辗转腾挪。

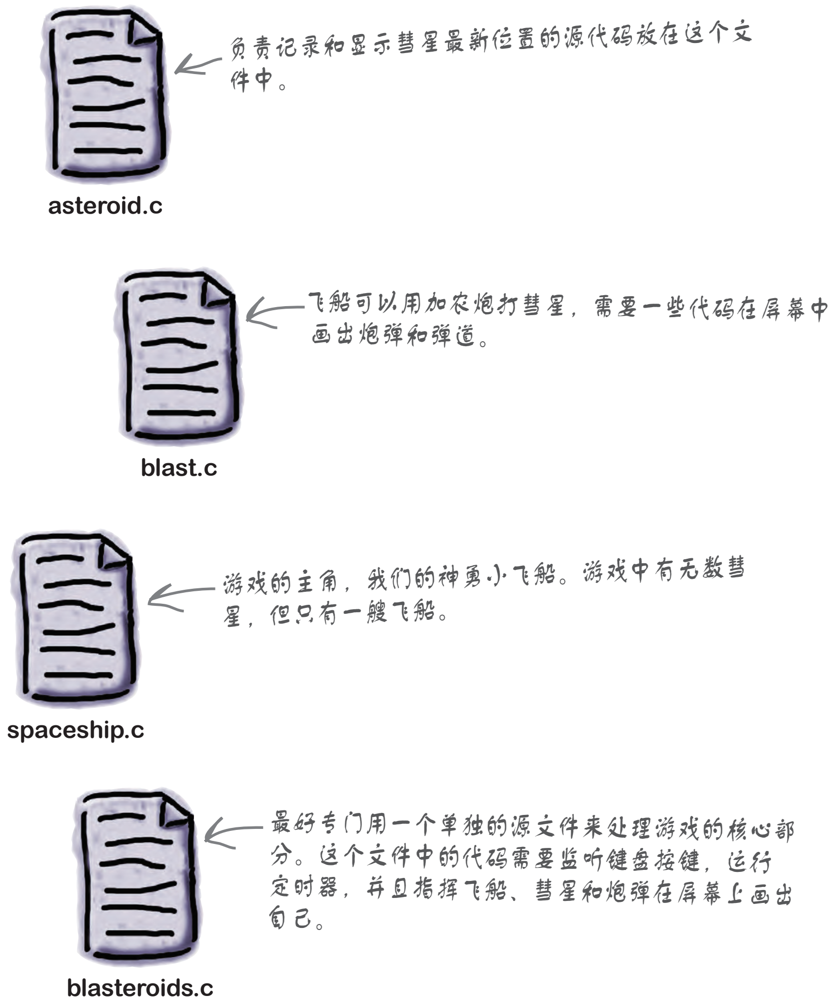
- **声音**

  Allegro有一个完整的声音库，有了它你就可以在游戏中加入声音。

构建游戏

你需要想好源代码按照什么方式来组织。绝大多数C程序员会把代码分成好几个源文件，这样不但能更快地重新编译游戏，而且可以一次处理更少的代码。分离代码让整个过程变得清晰明了。

分离代码的方法有很多，其中一种是为每个在游戏中显示的元素分别创建一个源文件：



宇宙飞船

当你要在屏幕中控制很多物体时，应该为每个物体创建一个结构。宇宙飞船的结构如下：

```
typedef struct {
  float sx;
  float sy;
  float heading;
  float speed;
  int gone;
  ALLEGRO_COLOR color;
} Spaceship;
```

它在屏幕中的坐标。

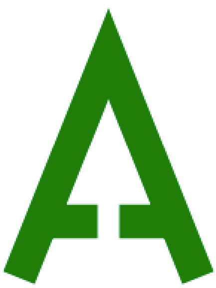
飞船的朝向。

是否阵亡？

飞船的外形

如果把代码设为以原点（稍后我们会讲）为参照绘制图形，就可以用以下代码画出飞船。

变量s是一个指向Spaceship结构的指针，我们把飞船漆成绿色。



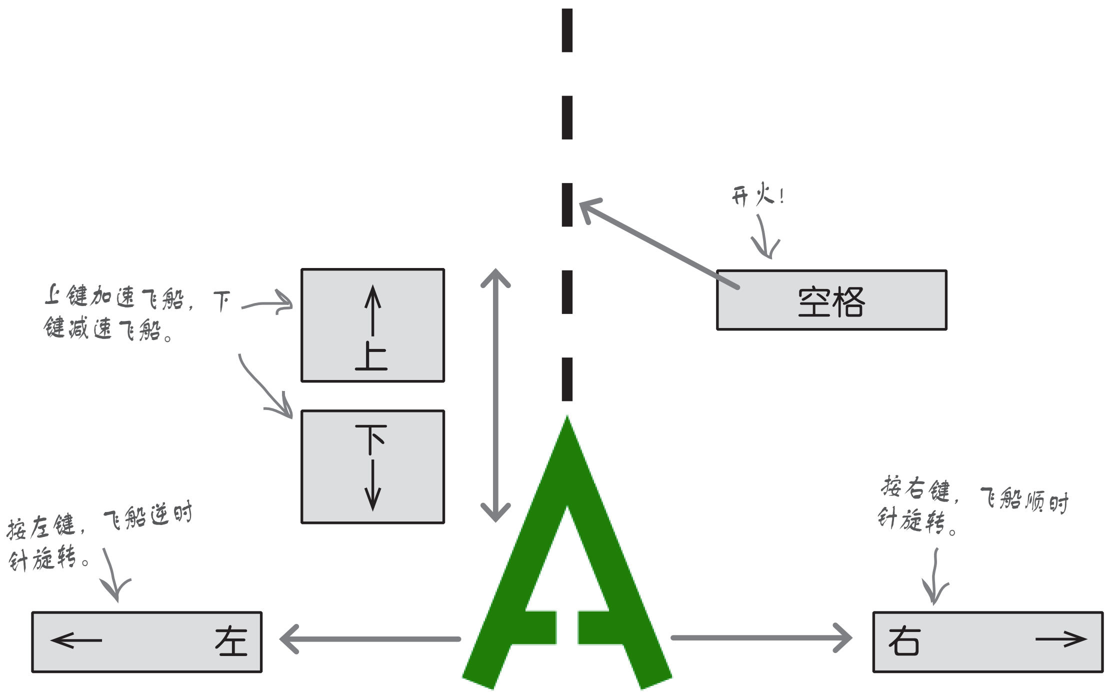
```
al_draw_line(-8, 9, 0, -11, s->color, 3.0f);
al_draw_line(0, -11, 8, 9, s->color, 3.0f);
al_draw_line(-6, 4, -1, 4, s->color, 3.0f);
al_draw_line(6, 4, 1, 4, s->color, 3.0f);
```

碰撞

飞船撞到彗星会立刻阵亡，玩家会少了一条命。飞船在创建后的五秒不检查碰撞，新飞船出现在屏幕的正中央。

飞船行为

游戏开始时飞船出现在屏幕的正中央。你可以让飞船响应键盘按键，在屏幕上移动：



不要让飞船加速得太快，最好不要超过每秒几百像素。飞船只能前进不能后退。

读取按键

世界上所有的计算机硬件几乎都用C语言来编程。但奇怪的是，居然没有用C语言读取按键的标准方式。所有标准函数（比如fgets()）都是在用户按下“回车”以后才读取按键。好在Allegro允许你实时读取按键。所有事件在发送到Allegro游戏前都会先进入一个队列，队列中的数据描述了用户按了哪个键、鼠标位置等信息。需要在代码中用一个循环来等待队列中出现事件。

就连getchar()函数也会先把字符缓存起来，直到你按下“回车”才读取。

```
ALLEGRO_EVENT_QUEUE *queue;
queue = al_create_event_queue();
ALLEGRO_EVENT event;
al_wait_for_event(queue, &event);
```

你创建了一个事件队列，像这样：

等待队列中的事件。

当你收到了一个事件，需要判断它是不是按键。可以通过读取它的类型来判断。

```
if (event.type == ALLEGRO_EVENT_KEY_DOWN) {
  switch(event.keyboard.keycode) {
  case ALLEGRO_KEY_LEFT:

    break;
  case ALLEGRO_KEY_RIGHT:

    break;
  case ALLEGRO_KEY_SPACE:

    break;
  }
}
```

飞船左转。

右转。

开炮！

炮弹

小样，让你尝尝炮弹的滋味！飞船上的加农炮可以发出炮弹，你的任务是画出弹道。下面是炮弹的结构：

```
typedef struct {
  float sx;
  float sy;
  float heading;
  float speed;
  int gone;
  ALLEGRO_COLOR color;
} Blast;
```

弹道

弹道是一条虚线，如果玩家射得很快，弹道就会密集起来，虚线就变成了实线，看起来就像加大了火力。

炮弹行为

不像游戏中的其他物体，炮弹从屏幕上消失以后不会再出现，也就是说你需要写一些创建炮弹和销毁炮弹的代码。飞船朝哪个方向飞炮弹就顺着哪个方向沿直线射出。炮弹的速率恒定，比如是飞船最快移动速度的三倍。被炮弹击中的彗星会一分为二。

彗星

彗星结构如下：

在屏幕中的位置

朝向哪个方向

当前的角度

转速

用来改变彗星大小的伸缩因子

是否毁灭？

```
typedef struct {
  float sx;
  float sy;
  float heading;
  float twist;
  float speed;
  float rot_velocity;
  float scale;
  int gone;
  ALLEGRO_COLOR color;
} Asteroid;
```

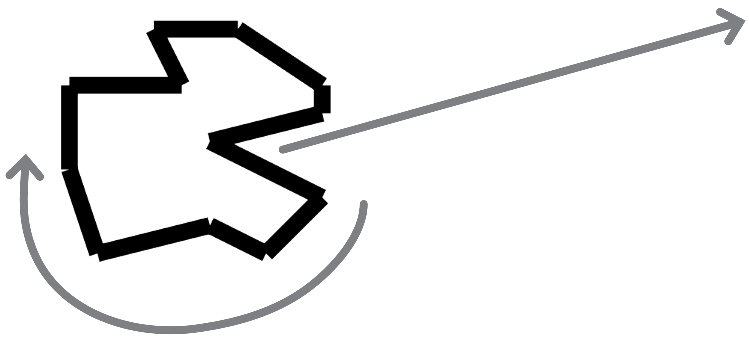
彗星的外形

下面这段代码以原点为参照画出彗星：

```
al_draw_line(-20, 20, -25, 5, a->color, 2.0f);
al_draw_line(-25, 5, -25, -10, a->color, 2.0f);
al_draw_line(-25, -10, -5, -10, a->color, 2.0f);
al_draw_line(-5, -10, -10, -20, a->color, 2.0f);
al_draw_line(-10, -20, 5, -20, a->color, 2.0f);
al_draw_line(5, -20, 20, -10, a->color, 2.0f);
al_draw_line(20, -10, 20, -5, a->color, 2.0f);
al_draw_line(20, -5, 0, 0, a->color, 2.0f);
al_draw_line(0, 0, 20, 10, a->color, 2.0f);
al_draw_line(20, 10, 10, 20, a->color, 2.0f);
al_draw_line(10, 20, 0, 15, a->color, 2.0f);
al_draw_line(0, 15, -20, 20, a->color, 2.0f);
```

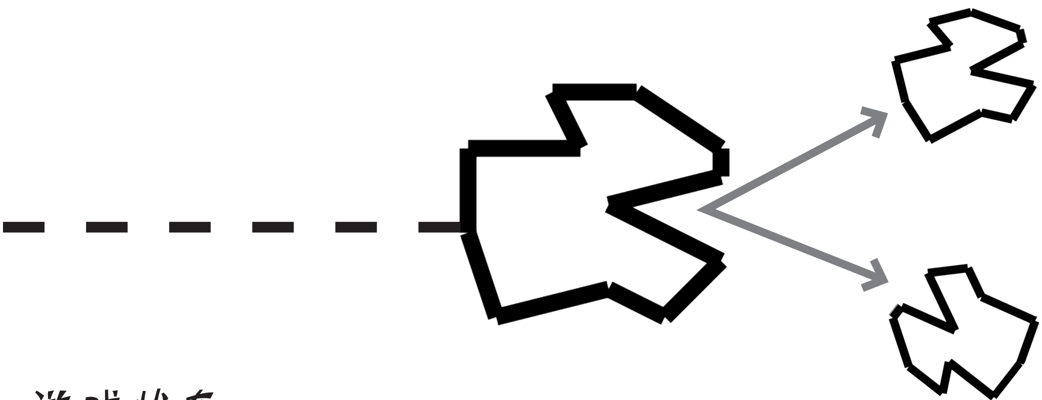
彗星

彗星在屏幕中沿直线移动，并绕着中心不断旋转。如果彗星从屏幕一侧飞出，马上会在屏幕另一侧出现。



命中彗星

如果加农炮发出的炮弹打中彗星，彗星就马上分成两瓣，每一瓣的大小是原来的二分之一。多次命中后彗星就会从屏幕上消失。每次命中彗星，玩家可以加100分。你认为应该用哪种数据结构保存屏幕上的彗星，一个很大的数组还是链表？



游戏状态

你还需要在屏幕上显示一些东西：玩家还剩几条命和当前得分。命用完以后，你需要友好地在屏幕的中央显示“游戏结束！”四个大字。

用“变换”移动物体

你要让物体在屏幕上“动起来”。飞船需要飞行，彗星需要旋转、漂移和变换大小。旋转、平移和伸缩操作需要很多数学知识，为此Allegro内置了一批“变换”函数。

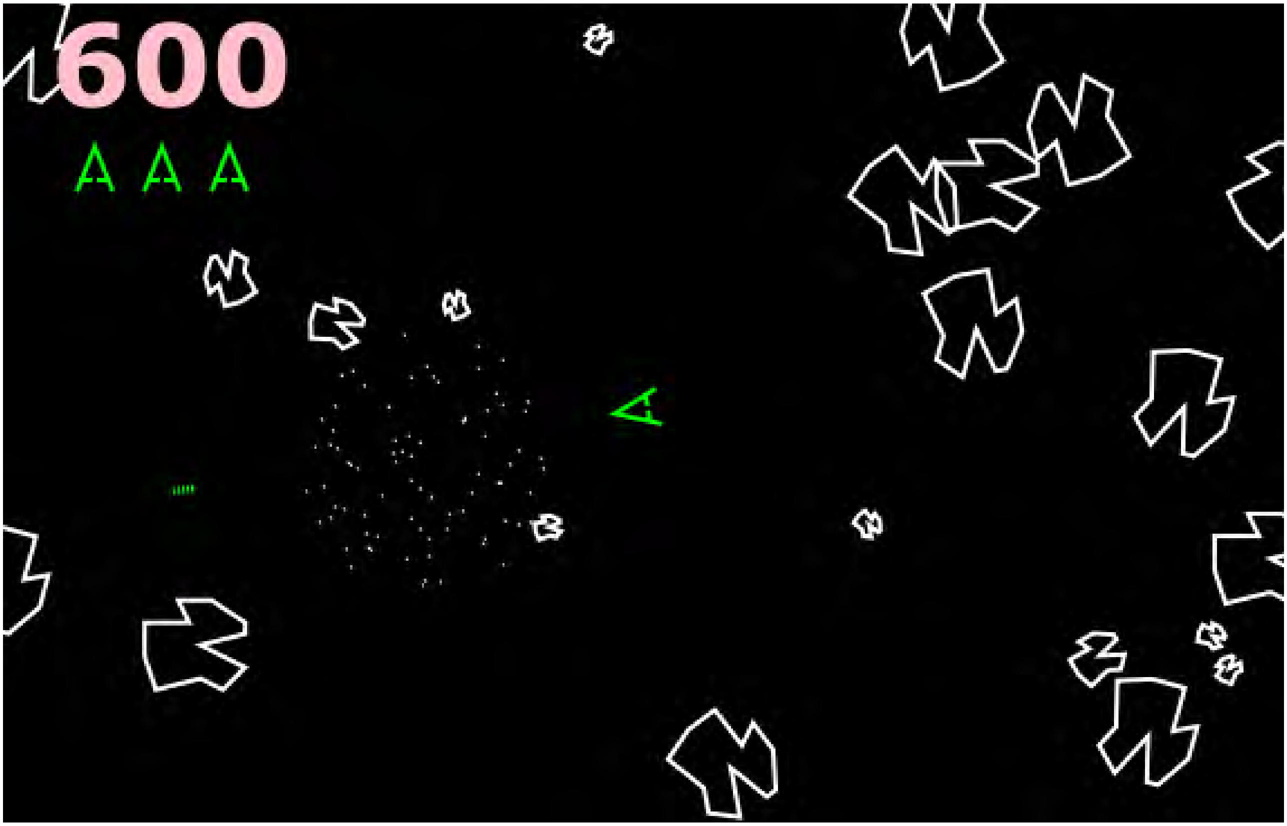
你在绘制物体时只需要以原点为参照画出它就行了。原点位于屏幕的左上角，坐标为(0, 0)，横着的是X轴，竖着的是Y轴。假设你要绘制飞船，可以先用变换函数把原点移动到飞船将在屏幕中出现的位置，然后根据飞船的旋转角度旋转原点，最后在原点处画出飞船即可。

可以像这样在屏幕上绘制飞船：

```
void draw_ship(Spaceship* s)
{
  ALLEGRO_TRANSFORM transform;
  al_identity_transform(&transform);
  al_rotate_transform(&transform, DEGREES(s->heading));
  al_translate_transform(&transform, s->sx, s->sy);
  al_use_transform(&transform);
  al_draw_line(-8, 9, 0, -11, s->color, 3.0f);
  al_draw_line(0, -11, 8, 9, s->color, 3.0f);
  al_draw_line(-6, 4, -1, 4, s->color, 3.0f);
  al_draw_line(6, 4, 1, 4, s->color, 3.0f);
}
```

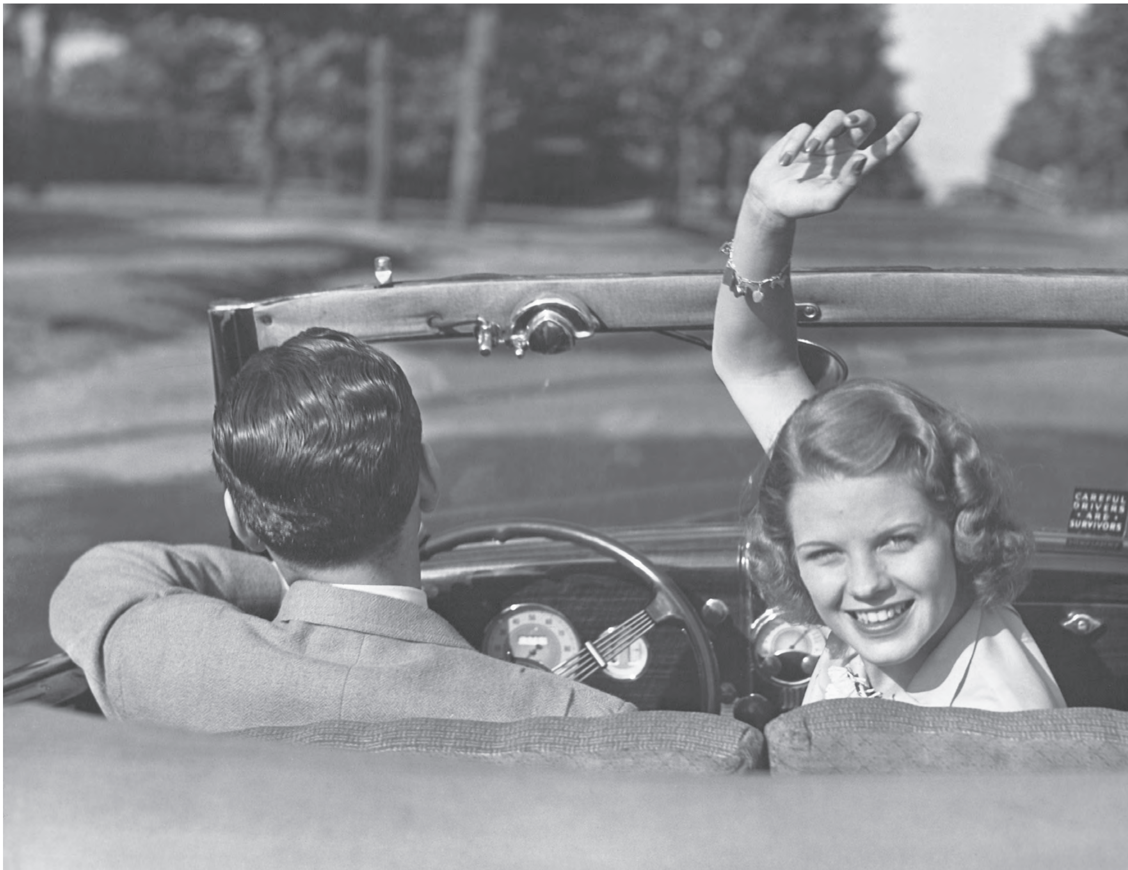
《爆破彗星》下线

全部写完之后，你就可以重温《爆破彗星》这款经典！



游戏中还有很多地方可以改进。比如你可以使用OpenCV。欢迎你把实验结果反馈给Head First实验室。

出山……



送君千里终须一别！

虽然我们舍不得你，但又不得不和你说再见。正所谓“纸上得来终觉浅，绝知此事要躬行”。我们在本书的附录中放了一些“美味佳肴”和一个建议阅读列表，等你看完以后就可以出山把所有知识运用到实践中。我们希望得知你的进展，欢迎在Head First实验室的网站www.headfirstlab.com上给我们留言，让我们知道C语言对你的帮助有多大！

十大遗漏知识点

革命尚未成功，同志还需努力。

我们认为你还需要知道一些事，如果不讲，总觉得哪里不对劲，但我们又不希望这本书重得只有大力士才提得动，所以我们只做简单介绍。在你放下这本书前，尽情地享用这些“美味佳肴”吧。

#1. 运算符

我们在本书中使用了一些运算符，例如基本的算术运算符+、-、*和/。C语言中还有很多其他运算符，它们可以让你生活得更简单。

递增与递减

递增将数字加1，递减将数字减1。这两种运算在C代码中出镜率很高，经常用来在循环中增减计数器的值，为此C语言提供了四个简单的表达式来简化这两种运算：

递增1，返回新值。→ **++i**

递增1，返回旧值。→ **i++**

递减1，返回新值。→ **--i**

递减1，返回旧值。→ **i--**

这些表达式都会改变i的值，++和--的位置决定了表达式返回i的原始值还是新值，例如：

```
int i = 3;
int j = i++;
```

← 这行代码执行以后，j==3，i==4。

三目运算符

如果想在条件为真时返回某个值，而在条件为假时返回另一个值，怎么做？

```
if (x == 1)
  return 2;
else
  return 3;
```

C语言中有一个三目运算符可以把以上代码压缩成一行：

```
return (x == 1) ? 2 : 3;
```

先是条件

然后是条件为真时表达式的值

最后是条件为假时表达式的值

位运算

C语言可以用来编写底层代码，为此它提供了一组位运算符：

运算符	说明
~a	a中所有位都取反
a & b	a中的位“与”b中的位
a | b	a中的位“或”b中的位
a ^ b	a中的位“异或”b中的位
<<	位左移（值增加）
>>	位右移（值减小）

<<运算符可以用来快速地将某个整型值乘以2的幂，但小心千万别溢出。

用逗号分割表达式

for循环在每次循环的末尾执行代码：

```
for (i = 0; i < 10; i++)
```

每次循环的末尾都会出现递增操作。

但如果你想在循环末尾执行多个运算怎么办？可以使用逗号运算符：

```
for (i = 0; i < 10; i++, j++)
```

递增i和j。

之所以要有逗号运算符是因为有时你不想用分号来分割表达式。

#2. 预处理指令

每次你在编译一个包含头文件的程序时都使用了预处理指令：

```
#include <stdio.h>   ← 预处理指令
```

预处理器会扫描C源文件然后生成一个修改过的版本，编译器会使用这个修改后的文件编译程序。对#include这条指令来说，预处理器会插入*stdio.h*文件的内容。指令总是出现在行首，以井号（#）字符开头。除了#include，用得最多的指令就是#define：

```
#define DAYS_OF_THE_WEEK 7
...
printf("一星期有%i天\n", DAYS_OF_THE_WEEK);
```

#define指令创建了一个宏，预处理器会扫描整个C源文件然后把宏的名字替换为它的值。宏不是变量，因为它的值在运行时无法改变。宏在程序编译前就被替换掉了，你甚至可以创建功能类似函数的宏：

```
                          x是宏的参数。
#define ADD_ONE(x) ((x) + 1)   ← 要注意在宏中加括号。
...
printf("答案是 %i\n", ADD_ONE(3));   ← 将输出"答案是4"。
```

在程序编译前，预处理器会用((3) + 1)替换ADD_ONE(3)。

条件编译

你还可以用预处理器来实现条件编译。条件编译可以开、关部分源代码：

```
#ifdef SPANISH   ← 如果SPANISH这个宏存在
char *greeting = "Hola";   ← ……就包含这段代码。.
#else
char *greeting = "Hello";   ← 否则就包含这段。
#endif
```

SPANISH宏定义与否会改变这段代码的编译方式。

#3. static关键字

想象你要创建一个带有计数功能的函数，可以这么写：

```
int count = 0;   ← 用来记录调用的次数
int counter()
{
  return ++count;   ← 每次都递增count
}
```

这段代码有什么问题吗？它使用了一个叫count的全局变量。因为count在全局作用域，所以其他函数可以修改它的值。如果你在写一个大型程序，就需要小心控制全局变量的个数，因为它们可能导致代码出错。好在C语言允许你创建只能在函数局部作用域访问的全局变量：

```
int counter()
{
  static int count = 0;
  return ++count;
}
```

static关键字表示将在两次counter()函数调用期间保持该变量的值。

虽然count是全局变量，但它只能在函数内部访问。

static关键字会把变量保存在存储器中的全局量区，但是当其他函数试图访问count变量时编译器会抛出错误。

用static定义私有变量或函数

也可以在函数外使用static关键字，它表示“只有这个*.c*文件中的代码能使用这个变量（或函数）”。例如：

```
static int days = 365;   ← 只能在这个源文件中使用这个变量。

static void update_account(int x) {   ← 只能在这个源文件中使用这个函数。
...
}
```

static关键字用来控制变量或函数的作用域，防止其他代码以意想不到的方式访问你的数据或函数。

#4. 数据类型的大小

你已经知道了怎么用sizeof运算符来查看数据类型在存储器中的大小，但如果你想知道数据类保存的值的范围呢？例如，你知道int在你的机器上占4字节，但int变量能保存的最大正数和最小负数分别是多少呢？理论上可以通过它占用的字节数计算出来，但这很麻烦。

为此可以使用定义在*limits.h*头文件中的宏。如果你想知道long可以保存的最大值，可以使用LONG_MAX宏。short可以保存的最小负数呢？用SHRT_MIN。下面这个例子显示了int和short的范围：

```
#include <stdio.h>
#include <limits.h>

int main()
{
  printf("On this machine an int takes up %lu bytes\n", sizeof(int));
  printf("And ints can store values from %i to %i\n", INT_MIN, INT_MAX);
  printf("And shorts can store values from %i to %i\n", SHRT_MIN, SHRT_MAX);
  return 0;
}
```

```
File Edit Window Help HowBigIsBig
On this machine an int takes up 4 bytes
And ints can store values from -2147483648 to 2147483647
And shorts can store values from -32768 to 32767
```

宏的名字取自数据类型：INT(int)，SHRT(short)，LONG(long)，CHAR(char)，FLT(float)和DBL(double)。你可以在它们后面加上_MAX（最大正数）或_MIN（最小负数）。如果想查看更具体的数据类型，还可以加上前缀U(unsigned)、S(signed)或L(long)。

#5.自动化测试

测试代码的重要性不言而喻，如果能把测试的过程自动化，生活会变得更轻松。几乎所有程序员现在都在使用自动化测试，C语言的测试框架也不胜枚举，而在Head First实验室最受欢迎的是AceUnit：

http://aceunit.sourceforge.net/projects/aceunit

AceUnit和其他语言中的*x*Unit框架很像（比如NUnit和JUnit）。

如果你写的是命令行工具，用的是Unix的命令行，还有一个好用的工具叫**`shunit2`**。

http://code.google.com/p/shunit2/

`shunit2`允许创建shell脚本来测试脚本和命令。

#6. 再谈gcc

本书你都在用gcc，但只使用了gcc最基本的功能，其实它可以做更多的事。gcc就像一把瑞士军刀，它有很多特性，利用这些特性你可以严格控制它生成的代码。

优化

gcc为了提高代码的性能会做很多工作。如果gcc发现你在循环中对一个变量赋了相同的值，它就会把赋值语句移动到循环外；如果一个小函数只在少数几个地方用到了，gcc就会把它转化为内联代码，然后插到程序中。

虽然gcc可以做很多优化，但绝大多数优化选项默认是关闭的。为什么？因为优化需要花很长时间，如果你尚处于开发阶段，通常希望快速编译代码。一旦准备发布代码，就可以打开优化选项。gcc一共有四个级别的优化：

标志	描述
-O	如果在gcc命令中加上-O（字母O）标志，就能得到第一级别的优化。
-O2	如果想提升优化等级，降低编译速度，就选择-O2。
-O3	如果想再升一级，就选-O3，它会使用-O和-O2中的所有优化，再附加一些额外的优化。
-Ofast	-Ofast会打开最高级别的优化，同时编译速度也会降到最低。谨慎使用-Ofast，因为它生成的代码可能和C标准相去甚远。

警告

如果代码没有严重错误，但做了一些可疑的事情，比如把一个类型的值赋给一个错误类型的变量，编译器就会显示警告。你可以用`-Wall`选项提高警告检查的门槛：

```
gcc fred.c -Wall -o fred
```

`-Wall`选项表示“所有警告（All Warnings）”，但因为一些历史原因，`-Wall`其实并不会显示所有的警告。如果你想让`gcc`那么做，就必须加上**`-Wextra`**选项：

```
gcc fred.c -Wall -Wextra -o fred
```

如果你希望遵循严格的编译，就可以使用**`-Werror`**选项，只要有一个警告，编译就会失败：

```
gcc fred.c -Werror -o fred
```

它表示“把警告当错误处理”。

当多人开发同一个项目时，`-Werror`就显得特别有用，因为它可以维持代码的质量。

更多`gcc`选项请参阅：

http://gcc.gnu.org/onlinedocs/gcc

#7. 再谈make

make是构建C程序的强大工具，但在本书中你只使用过一些简单的命令。为了看到更多make神奇的功能，请阅读Robert Mecklenburg的《GNU Make项目管理》：

http://shop.oreilly.com/product/9780596006105.do

这里先列举make的一些特性。

变量

变量可以大大缩短你的makefile，例如你想把一组标准的命令行选项传给gcc，就可以把它们定义成变量:

```
CFLAGS = -Wall -Wextra -v

fred: fred.c
  gcc fred.c $(CFLAGS) -o fred
```

可以用等号（=）定义变量，然后用$(...)读取变量的值。

使用%、^和@

很多编译命令看起来都很像：

```
fred: fred.c
gcc fred.c -Wall -o fred
```

这时你可以用%符号写一条更通用的“目标/生成方法”：

假设你想根据<文件>.c创建<文件>。

```
%: %.c
gcc $^ -Wall -o $@
```

#^是依赖项的值（.c文件）。

#@是目标的名字。

这些符号看起来有些奇怪。假设你想创建一个叫*fred*的文件，这条规则会让make去寻找一个叫*fred.c*的文件，然后生成方法会运行一条gcc命令，用依赖项（由特殊符号$^给出）创建目标fred（由$@给出）。

隐式规则

make工具对编译过程一清二楚，即使你不告诉它如何构建文件，它也可以使用隐式规则自行构建。例如，你有一个叫*fred.c*的文件，但没有makefile，可以用以下命令编译它：

File Edit Window Help MakeMyDay

```
> make fred
cc fred.c -o fred
```

cc是gcc的另一个名字。

即使我们不告诉make怎么编译，它同样创建了编译命令。

这是一条隐式规则。

原因是make内置了一批生成方法。关于make的更多信息，请参见：

http://www.gnu.org/software/make/

#8. 开发工具

当你在写C代码时，八成会对性能和稳定性有很高要求。如果你用gcc编译代码，很有可能对以下这些GNU工具感兴趣：

gdb

gdb（GNU Project Debugger，GNU调试器）允许你在程序运行期间研究它的代码。如果你想找出代码中隐蔽的错误，会发现它特别有用。gdb既可以在命令行中使用，也可在Xcode或Guile那样的IDE中使用。

http://sourceware.org/gdb/download/onlinedocs/gdb/index.html

gprof

如果你的程序没有预期的那么快，就有必要分析一下它的性能。gprof（GNU Profiler，GNU分析器）可以告诉你程序中哪个部分是最慢的，这样你就能进行适当优化。gprof会修改程序，修改后的程序在结束时会生成一份性能报告，然后你可以用gprof命令行工具分析它，找到程序的瓶颈所在。

http://sourceware.org/binutils/docs/gprof

gcov

还有一个分析工具叫gcov（GNU Coverage，GNU覆盖率测试工具）。gprof用来检查你代码的性能，而gcov用来检查代码中哪些部分运行了，哪些部分没运行。这在写自动化测试时特别有用，因为你需要保证测试代码覆盖了所有你想覆盖的代码。

http://gcc.gnu.org/onlinedocs/gcc/Gcov.html

#9. 创建GUI

你在本书前12章中都没有创建GUI程序，而在实验室中用Allegro和OpenCV库写过两个显示简易窗口的程序。在不同的操作系统中，GUI的创建方式有着天壤之别。

Linux——GTK

Linux有很多库可以用来创建GUI程序，其中最有名的要属GTK+（GIMP toolkit，GIMP工具包）：

http://www.gtk.org/

GTK+常用于Linux程序中，但你也可以在Windows和Mac中使用它。

Windows

Windows自带了十分高级的GUI库。Windows编程是非常专业的领域，在你开始创建GUI程序前，可能需要花一点时间来学习Windows API（Application Programming Interface，应用程序编程接口）。越来越多Windows程序开始用基于C的语言来开发，例如C#和C++。以下是Windows编程的在线介绍：

http://www.winprog.org/tutorial/

Mac——Carbon

苹果的GUI系统叫Aqua。如果你想在Mac上用C语言写GUI程序，可以用Carbon库，不过更时髦的方式是用Cocoa库，它需要用C语言的另一个后代Objective-C来编程。现在你已经来到了本书的终点，正是学习Objective-C的大好时机，Head First实验室的人对“书呆子牧场”出品的Mac编程书籍和课程爱不释手：

http://www.bignerdranch.com/

#10. 参考资料

以下是一些热门的C编程书籍和网站。

《C程序设计语言》
Brian W. Kernighan，Dennis M. Ritchie著
C语言的开山之作， C程序员应该人手一本。

《C语言参考手册》
Samuel P. Harbison，Guy L. Steele Jr.著
很好的C语言参考书，你在写代码时一定希望边上放着这本书。

《C专家编程》
Peter van der Linden著
如果你想了解更多高级C语言编程技巧就去看Peter van der Linden的这部佳作。

《实用C语言编程》
Steve Oualline著
这本书列出了一些实用的C语言开发细节。

网站

关于C标准：
http://pubs.opengroup.org/onlinepubs/9699919799/

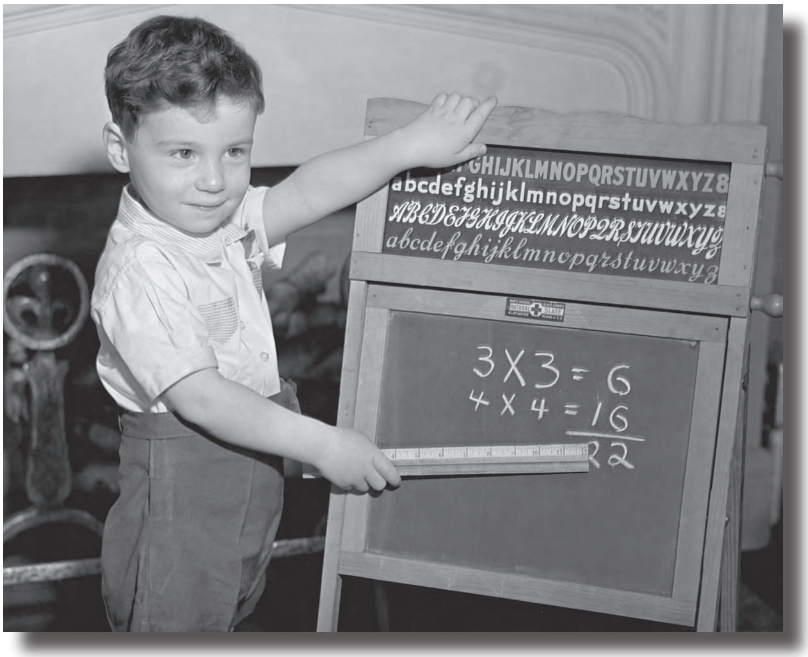
更多C教程：
http://www.cprogramming.com/

综合参考资料：
http://www.cppreference.com/

综合C编程教程：
http://www.crasseux.com/books/ctutorial/

ii 话题汇总

总复习



将C语言的特性尽收眼底。

我们把本书中出现过的所有关于C语言的话题和原理都汇总在了这里。把它们过一遍，看你还记得多少。每条话题都标明了来源章节号。如果你想不起来，很容易就能找到原文，甚至还可以把它们剪下来贴在墙上。

入门

第1章

简单的语句就是命令。

第1章

块语句被{和}包围。

第1章

如果条件为真，if语句就会运行代码。

第1章

switch语句高效地检查了一个变量的多种取值。

第1章

可以用&&和||把多个条件组合在一起。

第1章

每个程序都需要一个main()函数。

第1章

#include将外部代码（如用来输入输出的代码）包含进来。

第1章

源文件的文件名应该以.c结尾。

第1章

需要在运行之前先编译C程序。

第1章

gcc是最流行的C编译器。

第1章

可以在命令行中用&&操作符在编译之后马上运行程序，前提是必须编译成功。

第1章

-o指定了输出文件。

第1章

count++表示计数加1。

第1章

count--表示计数减1。

第1章

只要条件为真，while就会重复执行代码。

第1章

do-while至少执行一次代码。

第1章

用for写循环更简洁。

存储器和指针

第2章

scanf("%i", &x)可以让用户直接输入数字x。

第2章

用字符串初始化数组，数组会复制字符串中的内容。

第2章

&x返回x的地址。

第2章

用*a读取地址a中的内容。

第2章

局部变量保存在栈上。

第2章

char指针变量x可以这么声明：char *x。

第2章

&x称为指向x的指针。

第2章

数组变量可以用作指针。

第2章

可以简单地用fgets(buf, size, stdin)输入文本。

字符串

第2章 字符串字面值保存在只读存储器中。

第2.5章 字符串数组是数组的数组。

第2.5章 strstr(a, b)可以返回字符串b在字符串a中的地址。

第2.5章 strcat()可以连接字符串。

第2.5章 strcpy()可以复制字符串。

第2.5章 string.h头文件包含了有用的字符串处理函数。

第2.5章 可以用char strings[...][...]创建数组的数组。

第2.5章 strcmp()可以比较字符串。

第2.5章 strchr()用来在字符串中找到某个字符的位置。

第2.5章 strlen()可以得到字符串的长度。

数据流

第3章

printf()和scanf()使用标准输出和标准输入来交互。

第3章

标准输出默认在显示器上显示数据。

第3章

标准输入默认从键盘读取数据。

第3章

可以用重定向把标准输入、标准输出和标准错误连接到其他地方。

第3章

标准错误专门用来输出错误消息。

第3章

可以用fprintf(stderr,...)把数据打印到标准错误。

第3章

可以用fopen("文件名",模式)创建你自己的数据流。

第3章

三种模式分别是w(写入)、r(读取)、a(追加)。

第3章

命令行参数以字符串指针数组的形式传递给main()。

第3章

用getopt()函数读取命令行选项很方便。

数据类型

第4章

char是数值。

第4章

大整数用long。

第4章

小整数用short。

第4章

普通整数用int。

第2章

不同计算机的int的大小不同。

第4章

一般的浮点数用float。

第4章

高精度的浮点数用double。

多个文件

第4章

函数的声明与定义分离。

第4章

把声明放在头文件中。

第4章

用#include<>包含标准库头文件。

第4章

用#include“”包含本地头文件。

第4章

把目标代码保存到文件中，提高构建速度。

第4章

使用make管理代码构建过程。

结构

第5章 结构把数据类型组合在一起。

第5章 可以用“点表示法”读取结构中的字段

第5章 可以像初始化数组那样初始化结构。

第5章 有了“->”表示法，就能用结构指针更新字段，十分方便。

第5章 可以用typedef为数据类型创建别名。

第5章 可以用“指定初始化器”按名设置结构或联合的字段。

联合和位字段

第5章

联合可以在同一个存储器单元中保存不同的数据类型。

第5章

可以用枚举创建一组符号。

第5章

可以用位字段控制结构中的某些位。

数据结构

第6章

动态数据结构使用递归结构。

第6章

递归结构包含一个或多个链向相同结构数据的链接。

第6章

链表是动态数据结构。

第6章

在链表中插入数据很方便。

第6章

链表比数组更容易扩展。

动态存储

第6章

栈用来保存局部变量。

第6章

与栈不同，堆存储器不会自动释放。

第6章

malloc()在堆上分配存储器。

第6章

free()释放堆上的存储器。

第6章

strdup()会在把字符串复制到堆上。

第6章

存储器泄漏是指存储器分配出去以后，你再也访问不到。

第6章

valgrind可以帮助追踪存储器泄漏。

高级函数

第7章 有了函数指针，就可以把函数当数据传递。

第7章 函数指针是唯一不需要加*和&运算符的指针……

第7章 每个函数的名字都是一个指向函数的指针。

第7章 qsort()会排序数组。

第7章 排序函数接收比较器函数指针。

第7章 比较器函数决定如何排序两条数据。

第7章 有了函数指针数组，就可以根据不同类型的数据运行不同的函数。

第7章

参数数量可变的函数叫做“可变参数函数”。

第7章

包含stdarg.h，就可以创建可变参数函数。

静态库与动态库

第8章

#include<>会查找包括/usr/include在内的标准目录。

第8章

-L<路径名>在标准lib目录列表中添加目录。

第8章

-l<库名>会链接标准目录（例如/usr/lib）下的文件。

第8章

-I<路径名>在标准include目录列表中添加目录。

第8章

ar命令创建目标文件的存档。

第8章

库存档名形如libXXX.a。

第8章

库存档是静态链接的。

第8章

gcc -shared把目标文件转化为动态库。

第8章
动态库在运行时链接。

第8章
动态库在不同的操作系统上有不同的名字。

第8章
动态库的后缀名有.so、.dylib、.dll和.dll.a等。

进程间通信

第9章

system()会把字符串当成命令运行。

第9章

fork()复制当前进程。

第9章

fork()+exec()创建子进程。

第9章

execl() = 参数列表

execle() = 参数列表 + 环境变量

execlp() = 参数列表 + 搜索PATH

execv() = 参数数组

execve() = 参数数组 + 环境变量

execvp() = 参数数组 + 搜索PATH

第10章

进程可以用管道通信。

第10章

pipe()创建通信管道。

第10章

exit()立即终止程序。

第10章

waitpid()等待进程结束。

第10章

fileno()查找描述符。

第10章

dup2()复制数据流。

第10章

信号是O/S发出的消息。

第10章

用sigaction()处理信号。

第10章

程序可以用raise()向自己发送信号。

第10章

alarm()会在几秒钟以后向进程发送SIGALRM信号。

第10章

kill命令发送信号。

第12章

普通进程一次只做一件事。

网络与套接字

第11章

Telnet是一个简易网络客户端。

第11章

用socket()函数创建套接字。

第11章

服务器BLAB四部曲：

B = bind()

L = listen()

A = accept()

B = 开始对话

第11章

用fork()克隆子进程，同时处理多个客户端。

第11章

DNS=Domain name system（域名系统）

第11章

getaddrinfo()根据域名找地址。

线程

第12章

有了线程，进程一次就能做多件事。

第12章

线程是“轻量级进程”。

第12章

POSIX线程(pthread)是一个线程库。

第12章

pthread_create()创建线程来运行函数。

第12章

pthread_join()会等待线程结束。

第12章

线程共享相同的全局变量。

第12章

如果线程读取并更新了相同变量，代码的运行结果将不可预测。

第12章

互斥锁是用来保护共享数据的锁。

第12章

pthread_mutex_lock()在代码中创建互斥锁。

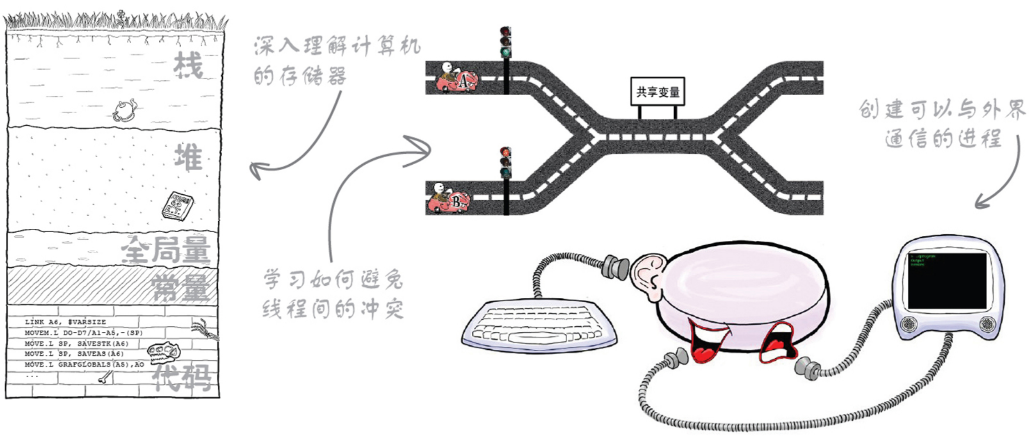
第12章

pthread_mutex_unlock()释放互斥锁。

编程语言/C

你能从这本书中学到什么？

你有没有想过可以轻松学习C语言？《嗨翻C语言》将会带给你一次这样的全新学习体验。本书贯以有趣的故事情节、生动形象的图片，以及不拘一格、丰富多样的练习和测试，时刻激励、吸引、启发你在解决问题的同时获取新的知识。你将在快乐的气氛中学习语言基础、指针和指针运算、动态存储器管理等核心主题，以及多线程和网络编程这些高级主题。在掌握语言的基本知识之后，你还将学习如何使用编译器、make工具和其他知识来解决实际问题。



这本书有什么特别之处？

《嗨翻C语言》运用认知科学和学习理论的最新成果，精心为你打造了一次多感官的学习体验，绝对能够嗨翻你的大脑，激发你的学习热情。它的特别之处是：

- 用图片等可视化手段，提高学习效率；
- 使用对话和有个性的叙述风格，讲故事而不是照本宣科；
- 调动读者左右半脑和各种感官，让学习者思考得更深入；
- 吸引并抓住读者的注意力，让学习新技术一点都不枯燥。

"《嗨翻C语言》可能很快就会被证明是学习C语言的最佳书籍。我觉得它会成为每所大学C语言的标准教材。很多编程书籍因循守旧。不过这本书却使用了完全不同的方式。它将教你如何成为一名真正的C程序员。"

——**Dave Kitabjian**,
NetCarrier Telecom
软件开发总监

"《嗨翻C语言》是一本用经典'Head First'的方式轻松介绍C语言的教材。图片、笑话、练习以及实践让读者逐渐并稳固地掌握C语言的基础知识……由此，读者可以进入Posix和Linux系统编程中更高级的技术殿堂。"

——**Vince Milner**,
软件工程师

封面设计：Karen Montgomery　张健
图灵社区：www.ituring.com.cn
新浪微博：@图灵教育　@图灵社区
反馈/投稿/推荐信箱：contact@turingbook.com
热线：(010)51095186转604

分类建议　计算机/编程语言/C语言

O'REILLY®
oreilly.com.cn
headfirstlabs.com

ISBN 978-7-115-31884-8
ISBN 978-7-115-31884-8
定价：99.00元

人民邮电出版社网址：www.ptpress.com.cn
O'Reilly Media, Inc.授权人民邮电出版社出版

此简体中文版仅限于中国大陆（不包含中国香港、澳门特别行政区和中国台湾地区）销售发行
This Authorized Edition for sale only in the territory of People's Republic of China (excluding Hong Kong, Macao and Taiwan)

欢迎加入

图灵社区

电子书发售平台

电子出版的时代已经来临，在许多出版界同行还在犹豫彷徨的时候，图灵社区已经采取实际行动拥抱这个出版业巨变。相比纸质书，电子书具有许多明显的优势。它不仅发布快，更新容易，而且尽可能采用了彩色图片（即使有的书纸质版是黑白印刷的）。读者还可以方便地进行搜索、剪贴、复制和打印。

图灵社区进一步把传统出版流程与电子出版业务紧密结合，目前已实现作译者网上交稿、编辑网上审稿、按章发布的电子出版模式。这种新的出版模式，我们称之为“敏捷出版”，它可以让读者以较快的速度了解到国外最新技术图书的内容，弥补以往翻译版技术书“出版即过时”的缺憾。同时，敏捷出版使得作、译、编、读的交流更为方便，可以提前消灭书稿中的错误，最大程度地保证图书出版的质量。

开放出版平台

图灵社区向读者开放在线写作功能，协助你实现自出版的梦想。你可以联合二三好友共同创作一部技术参考书，以免费或收费的形式提供给读者，这极大地降低了出版的门槛。成熟的书稿，有机会入选出版计划，同时出版纸质书。

图灵社区引进出版的外文图书，都将在立项后马上在社区公布。如果有意翻译哪本图书，欢迎来社区申请。只要通过试译的考验，即可签约成为图灵的译者。当然，要想成功地完成一本书的翻译工作，是需要有坚强的毅力的。

读者交流平台

在图灵社区，读者可以十分方便地写作文章、提交勘误、发表评论，以各种方式与作译者、编辑人员和其他读者进行交流互动。提交勘误还能够获赠社区银子。欢迎大家积极参与社区开展的访谈、审读、评选等多种活动，赢取银子，可以换书哦！